

هندسة البرمجيات

باستخدام لغة ADA

SOFTWARE ENGINEERING

With ADA

Grady [illegible]h

ترجمة و إعداد : د. درغام ميخائيل



دار الرضا للنشر

هندســة البرمجيــات

باستخدام لغة ADA

SOFTWARE ENGINEERING WITH

ADA

ترجمة وإعداد

د. درغـام ميخائيـل

سلسلة الرضا للمعلومات ٨٧

سلسلة علمية متميزة لنشر ثقافة الإدارة الحديثة والمعلوماتية بغية تطوير المؤسسات والشركات التي تسعى للريادة.

دار الرضا للنشر

تجهيز – قرب فندق برج الفردوس – هاتف: ٢٢٢٤٦١٧
تلفاكس: ٢٢٢٢١٦٣
ص.ب: ٤٢٦٧
E-mail: Reda-Center @ net.sy
Site:www/redapress.com

الإخراج: مركز جديدة للخدمات الطباعية – تلفاكس: ٦٨١٦٦٣٠

ايلول ٢٠٠٠

إهداء

سيادة رئيس الجمهورية العربية السورية

الفريق الدكتور بشار الأسد

أتقدم من سيادتكم بعملي المتواضع "هندسة البرمجيات باستخدام لغة ADA" عربون تقدير ووقار واحترام لمن رسخ قواعد المعلوماتية في بلدنا وقاد مسيرة الكرامة والعزة والتقدم والتطوير والتحديث استمراراً لنهج القائد العظيم الراحل الخالد الرئيس حافظ الأسد رحمه الله.

أملاً أن أبقى جندياً في مسيرتكم المظفرة وفياً لمسيرة البناء والتصحيح وعلى السير قدماً خلف قيادتكم الحكيمة حتى تتحقق أهداف أمتنا في النصر والتحرير.

الدكتور درغام ميخائيل

تقديم الناشر

دائماً نحلم بأن نجد لدينا صناعة برمجيات عربية مزدهرة ومنتجة، وهو حلم يعبر عن الرغبة في النجاح والتطور وتحقيق الذات، طالما أن هذه الصناعة تحتاج للعقول والمادة الرمادية في الدماغ، ولا تستهلك مواد أو ثروات طبيعية، وهذه الصناعة التي ارتبط ظهورها بتطورات العصر الحديث وتحدياته، وبالمساهمة في بناء مجتمع المعلومات العربي، ودعم انتشار اللغة العربية، والمساهمة في بناء مجتمع المعلومات والمعرفة، والذي يتطلب أقصى درجات التكثيف والإنتاجية المعرفية، تلك الصناعة التي تتنافس عليها أرقى الدول والتي لها منعكساً من خلال نجاح الهند في بناء صناعة برمجيات عالمية، تنافس بها أميركا واليابان وأوربا.

كانت وما زالت لغات البرمجة اليوم هي الأساس لبناء تطبيقات معلوماتية متنوعة وناجحة في عصر المعلومات الذي أصبحت فيه البرمجيات سلعة وصناعة من الصناعات الاستراتيجية التي تسعى كل الدول للمساهمة فيها، والبرمجة كمجموعة علوم رياضية منطقية تقنية تسعى في إنتاج برامج وتطبيقات تساهم في إعادة تنظيم أعمال يدوية وورقية بأسلوب مؤتمت يبني الأرشيف وقواعد البيانات في المجالات الإدارية، وبذلك يساعد في بناء كل أنواع البرامج والتطبيقات من الألعاب إلى برامج الخدمات إلى الأتمتة الإدارية وغيرها، والمهم أن لغات البرمجة كثيرة ولها أجيال عدة تطورت بنيوياً مع مراحل تطور البنية المادية للحاسوب، وتتدرج اللغات برقيها، حتى تصبح من اللغات البنيوية التي تستعمل تعابير باللغة الإنكليزية تقترب من اللغة المتداولة، وتتعامل مع العناصر والمكتبات البرمجية الجاهزة، كما تطورت تلك اللغة مع البيئة المرئية وأصبحت تتعامل مع مختلف عناصر المالتي ميديا، ولكن بعد أن نشأت لغات البرمجة التقليدية المختصة مثل كوبول وفورتران، وكانت التطبيقات والبرامج بسيطة ومحددة ظهرت أزمة البرمجيات وظهر الفساد البرمجي حين تعرضت البرمجيات في تعقيدها لمتاهات في التطوير، والعجز عن تلبية احتياجات الأتمتة الإدارية المطلوبة.

وفي السنوات الأخيرة نلاحظ أن لغات البرمجة قد تحولت إلى لغات برمجة جاهزة تحتوي مكتباتها على العديد من البرمجيات الجاهزة، التي توفر على المبرمج المهام التكرارية وتجعله يتعامل مع مكتبة برامج جاهزة، وهذا ما جعل الجهد الإبداعي في

البرامج والتطبيقات يتركز على هندسة البرمجيات، في تصميمها ودقة خدماتها وتفاصيلها المدروسة.

ومنذ ظهور أزمة البرمجيات وفسادها، ظهرت أهمية علوم هندسة البرمجيات، ذلك العلم الشامل والأعم من البرمجة الذي يساهم في تصميم البرامج وتخطيط وظائفها، وهو موضوع مشابه لأي موضوع هندسي يضع أساساً لطرق بناء البرنامج وتحقيقه للنتائج المطلوبة واستبعاد كل الحالات الخاصة التي تجعل البرنامج قاصراً أو عاجزاً، ويجعل للبرنامج خطة ومبادئ في التصميم تحمي البرنامج من أخطاء واحتمالات خاطئة تعيق الحصول على نتائج صحيحة. وتساهم في البناء المتكامل والمختبر للبرنامج، كما تساهم هندسية البرمجيات في تحليل عمل البرنامج والهدف منه واستعمال الخوارزميات أو الأدوات المناسبة للحصول على تطبيق ناجح، إن البرامج اليوم التي تقوم بها المؤسسات تكون مرنة ومحققة لنتائج لمختلف الشركات أو الجهات التي تحتاجها وترى فيها تنوعاً وإغناءً من خلال النسخ المختلفة للبرنامج التي تحدث دوماً وتدرس مزاياها بحيث يتم استبعاد الأخطاء وزيادة السرعة وتلبية احتياجات جديدة في المؤسسة أو الشركة في تطبيقاتها.

ونحن اليوم في هذا العصر المعلوماتي الذي نسعى فيه لبناء صناعة برامج وتطبيقات تساهم في أتمتة الأعمال الإدارية وبناء بنوك المعلومات بحاجة قصوى لهندسة البرمجيات التي تجعل برامجنا مدروسة وبعيدة عن الأخطاء وأكثر مرونة بالتعديل، وفيها بنية منطقية سريعة وجيدة تختلف عن أسلوب اعتماد المبرمجين وذلك الذي يوقع في أخطاء تصميمية قاتلة، وما أكثر البرمجيات التي استثمر في تطويرها الكثير وتم الاستغناء عنها في النهاية لأخطاء في التصميم أو لغياب المصمم الأساسي أو لكون المبرمج غير مختص في هندسة البرمجيات فتم إنجاز البرنامج بأسلوب خاص، فالبرمجة كلغات تبقى أدوات إنجاز أما هندسة البرمجيات فهي فكر منظم لعمل المبرمج تجعله يخطط لبرنامجه وينظم الأجزاء المختلفة لبرنامج ويربطها بشكل علمي، وخصوصاً في مجال قواعد البيانات، كما أن مهندس النظم يختبر برنامجه تجاه أي حالة خاصة تحمي المستثمر من أي أخطاء شائعة أو طارئة، أو متوقعة الحدوث، كما أن مجال هندسة البرمجيات هو موضوع تخطيطي للبرنامج يساهم في بناء نموذجي للبرنامج كوظائف وبيانات، ومن خلاله يقوم المبرمج بدراسة تحليلية لعمله، والوظائف والإمكانيات التي يقوم بها، وهذا ما يجعل تعديل

البرنامج مرتبطاً بتعديل تصميمه، كما يدرس تحليل الأعمال الواجب أتمتتها ويدرس عوامل أمان المعلومات وحمايتها، كما يدرس حجم البيانات ونموها، وهذا ما يجعل مجال هندسة البرمجيات أساسي لكل معلوماتي مبرمج يسعى للمشاركة في بناء برمجيات وتطبيقات ناجحة.

ولندرة المراجع باللغة العربية لهذا المجال العلمي الهام، كان نشر هذا المرجع من ترجمة وإعداد الدكتور درغام ميخائيل وهو الخبير في حياته العملية والعملية في شؤون تصميم البرمجيات وأهمية هندسة البرمجيات في نجاح تجارب الأتمتة ونجاح البرامج، وفي تحقيق خدماتها وتطويرها باستمرار، وهذا المجال العلمي الهام في عصر المعلومات وصناعة البرمجيات، يتطلب نشر مراجع عديدة وقد كان اختيارنا أن نبدأ هذا المجال الهام بدراسة هندسة باستخدام لغة آدا تلك اللغة العلمية المتميزة والمسماة باسم أول مبرمجة في العالم وهي المبرمجة آدا ابنة اللورد بايرون، وهذه اللغة المميزة التي اعتمدتها وزارة الدفاع الأميركية لتميزها وقوتها الكبيرة في هندسة البرمجيات.

هاني شحادة الخوري

دمشق ٢٠٠٠/٩/١

المحتويات Contents

الصفحة

الملحقات


Converted by Tiff Combine - (no stamps are applied by registered version)

تقديم Preface

هندسة البرمجيات مع ADA:

تعتبر ADA، لغة برمجة ذات استخدام عام، وذات قدرة تعبيرٍ عاليـة. وقـد طُـوِّرت بناء على مبادرة من وزارة الدفـاع الأمريكيـة، كجـوابٍ علـى أزمـة تطويـر البرمجيـات. وقـد صُممت خصيصاً لمجال الأنظمة المعلوماتية الضخمة، وللأنظمة المحمولة في الزمن الحقيقي، بالرغم من أن لها تأثيراً كبيراً في مجالات تطبيقية أخرى.

وبشـكلٍ مغـايرٍ لكافـة البرمجـة عاليـة المسـتوى، مثـل COBOL، FORTRAN، أو حتى PASCAL، فإن ADA لا تتضمن العديد من أسس البرمجـة والتطويرالحديثـة فحسـب بل تقدم أيضاً، الطرق للتحقق منها. وهكذا نحصل على أكبر ربح لهذا المجهود من أجل لغة مشتركة ذات مستوى عـال، عـن طريـق تطبيـق طـرق تطويـرٍ جيـدة، تـم تسـهيلها بواسـطة استخدام «آدا»ADA، كلغةٍ تسـمح بـالتعبير عنهـا. وبالنتيجـة، فـإن إدخـال ADA، يمثـل فرصةً ممتازةً لتحسين الدقة، والوثوقية، والفعالية، وقابلية صيانة النظم البرمجية.

مع ذلك، فإن ADAليست فقط لغة برمجة إضافية، ولكن يمكن استخدامها فيما هو أبعد من ذلك. فهي تقدم وسيلة في غاية الأهمية والقوة، إذا ما اقترنت ببيئة تطويـر برمجي، تساعد في فهم المسـائل وفـي التعبـير عـن حلولهـا، بطريقـةٍ تعكـس مباشـرةً تعددية الأبعاد للعالم الحقيقي.

الطبعة الثالثة من هندسة البرمجيات مع ADA:

في الحقيقة، لقد دخلت ADA الإتجاه السائد لعلم الحاسـوب. ولقـد كُتبـت الطبعـة الثانية من هذا الكتاب كجوابٍ على الإستخدام المتنامي للغات .وتمثل النسخة الثالثة هذه، بشكل عام، النسخة الثانية، مع بعض النقاط الإضافية. حيـث تـمّ ضـم، وكتابـة، و إعـادة تنظيم الفصول، لتعكس بشكل أفضل، طريقة التفكير الحالية. وتضم هذه الطبعة مقاطع ترميزٍ موسعة، مع برامج كاملة، مكتوبة بأسلوبٍ أكثر حداثـة. وبشـكلٍ أكـثر أهميـة، تؤكـد هـذه الطبعة، على استخدام تأثيرات ADA في مجال هندسة البرمجيات.

وهكذا، فإنّ هذا الكتاب، يعتبر مرجعاً كاملاً لـ ADA، الذي يناسب كلاً من المبرمج الذي يرغب بخلق نظم ADA، والمدير الذي يحتاج إلى فهم كيفية تطبيق هذه الوسيلة القوية. ويفترض هذا الكتاب، فهم المبادئ الأساسية لقواعد البرمجة.

الأهداف:

لا يعتبر هذا الكتاب مجرد مدخل إلى ADA. فلقد كُتِب لَيفي بثلاثة أهدافٍ، هي:

- تقديم دراسةٍ مكثفةٍ لميزات ADA.
- تقديم أمثلةٍ ممتعةٍ عن التصميم الجيد، و البرمجة بـ ADA.
- تقديم طريقة تطويرٍ غرضية التوجه، تستثمر قوة ADA، و تساعد بالإضافة لذلك، على إدارة النظم البرمجية الضخمة.

وبشكل مختصر، لايكتفي هذا الكتاب فقط، بوصف البرمجة بـ ADA بالتفصيل، لكنه يقترح أيضاً، طرقاً يتم من خلالها التطبيق الأفضل لميزات اللغة، من أجل خلق النظم البرمجية.

تمثيل المحتويات (Content Features):

البنية (Structure):

تقدم كثير من الكتب، تفاصيل من لغة البرمجة، فقط، من خلال منظورٍ دلالي أو قواعدي. وفي هذا الكتاب، بدأنا بطريقة تصميمٍ برمجية، و من ثمّ قدمنا ADA من الأعلى للأدنى، في سياق طرقٍ برمجيةٍ جيدة.

ولقد تمّ تقسيم الكتاب إلى ست حزم، تتألف كل واحدة منها من ثلاثة إلى أربعة فصول، مرتبطةٍ منطقياً. وتبدأ الحزمة الأولى بنظرةٍ موجزةٍ عن مجال مسألة ADA، وتحتوي على مسحٍ لتاريخ تطور ADA، بهدف توضيح بعض ميزات اللغة. ويتمثل الهدف الأساسي من هذه الحزمة، بمناقشة أساسيات هندسة البرمجيات، كما أنها ترتبط بالتطوير غرضي التوجه. وتنتهي الحزمة الأولى بنظرةٍ شاملةٍ على اللغة.

وتحتوي الحزمة الثانية، على مسألتي التصميم الأوليتين، من خمس مسائل تصميم (وإحدى هذه المسائل، تمّ الرجوع إليها، وتوسيعها مؤخراً). كما تحتوي

أيضاً، على مناقشةٍ لأنواع معطياتٍ مجردة، و تفاصيل عن أنواع المعطيات المقدمة في لغة ADA.

ومن الحزمة الثالثة وحتى الحزمة السادسة، يُطرَح تقديم مفصل لـ ADA، مبني حول خمسة أمثلة تصميمٍ مكتملة. والمسائل ذات تعقيدٍ متزايد، ومع بعض، تتطلب هذه المسائل تطبيق جميع ميزات ADA. بالإضافة لذلك، تُقدِّم هذه المسائل وسيلةً لإيضاح طريقة التطويرغرضية التوجه، بالإضافة إلى أسلوب برمجةٍ، يساعد على قابلية الفهم. وتعرض الفصول التي تضم هذه المسائل الخمسة الضخمة، مناقشةً مفصلةً لبُنى لغة ADA. وتضم الحزمة السادسة أيضاً، مناقشةً للمسائل المرتبطة بنظم البرمجة الضخمة جداً، و تعرض آخر ما استجد من مسائل التصميم.

الموارد (Resources):

خُتِمَ الكتاب بأربعة ملاحق، تقدم تفاصيل تقنية إضافية عن ADA. وقد تمّ ترتيب محارف الملاحق بإحكام، لتتراصف مع LRM «دليل المرجع اللغوي» (Language Reference Manual). وتصف الملاحق الثلاثة الأولى، العناصر المسبقة التعريف لهذه اللغة، وتوصيف جميع الحزم البرمجية المسبقة التعريف، بما في ذلك الحزم البرمجية المتعلقة بجميع هيئات الدخل/الخرج. والملحق التالي، يحتوي على الدليل الجيد و المفهوم، وهو سهل القراءة.

تنظيم الدروس (Course Organization):

يمكن استخدام هذا الكتاب، بعدة طرق. فقد دُرِّست المادة في فصل واحد (٤٠ ساعة درسية)، ولمدة أربعة أسابيع، كما يمكن أن تدرَّس في محاضرات على فترة خمسة أيام، وفق الجدول التمهيدي اللاحق.

بالإضافة لذلك، فإن البنية التالية، تسمح بتقديمٍ مختصرٍ لتطبيق ADA، لمديري البرامج:

الكتلة ١ الفصل ١ : مقدمة.

الكتلة ٢ الفصل ٢ : هندسة البرمجيات.

الفصل ١٩ : دورة حياة البرمجة مع ADA.

الكتلة ٣ الفصل ٣ : التصميم غرضي التوجه.

الفصل ٤ : لمحة عن اللغة.

الكتلة ٤ الفصل ٢١ : مسألة التصميم الخامسة: إظهار رأس مرتفع.

الفصل ٢٠ : البرمجة على نطاق واسع.

الوثوقية (Reliability):

تمّ إصدار كتاب: The Reference Manual for the ADA Programming Language، من قبل وزارة الدفاع الأمريكية كمعيارٍ عسكري، و من قبل المعهد الوطني الأمريكي ANSI/MIL-STD1815A في ١٩٨٣/٢/١٧. وإن جميع المواد الواردة في هذا الكتاب تتناسب مع المعيار. ولتأكيد دقة ذلك، فإن جميع أمثلة التصميم، وقطع الترميز في الكتاب، قد تمّ فحصها باستخدام مترجمٍ صحيحٍ لـ ADA.

فريق المؤلفين:

إن هذه النسخة، هي نتيجة لتعاون جهود ثلاثة أشخاص هم: Doug Bryan، Grady Booch، Charles G.Petersen. وقد أثّرت الجهود المشتركة لهؤلاء المؤلفين والأساتذة الثلاث، في عدة آلافٍ من المبرمجين و المدراء، خلال العقد الماضي.

فـ «غريدي بوش» Grady Booch، خبير بـ ADA معترف به. ولقد درس في أكاديمية القوى الجوية في الولايات المتحدة، وأدار مؤتمراتٍ في كل مكان من الولايات المتحدة، وفي أوروبا. ولقد قدّم التفاصيل التقنية عن اللغة، لمجموعاتٍ في عدة مستويات - طلاب جامعيين، طلاب دراسات عليا، غير مبرمجين، مبرمجين محترفين، ومديري برامج. وخلال هذه التجربة، قد تم اختبار طرقٍ متعددةٍ لعرض ميزات اللغة، ومراقبة نجاح وإخفاق الطرق، والإصغاء للإحتياجات الحقيقية لممارسة تطوير البرمجيات.

وأما Doug Bryan من جامعة Stanford، فقد أصبح إسما مألوفاً في جماعة ADA، واُعتبر"محامٍ لها". ولقد ربح هذه الشهرة، من خلال مقالٍ في صحيفة "Dear ADA"، و التي تمثل ميزاتٍ نظاميةٍ لحروف ADA، وتصدر كل شهرين، من قبل جمعية الحواسيب المهتمة خاصة بـ ADA (SIGADA).

وأمّا Charles G.Petersen، فهو مدرِّس في جامعة ولاية الميسيسبي، بدرجة مهندس في علم الحاسوب. فله خبرة ١٠ سنوات في الهندسة، وأكثر من ٢٠ سنة بتدريس التجربة على مستوى الجامعة. ولقد ظل يدرِّس ADA لمدة ١٠ سنوات، ونشر ثلاث كتب نصية عن ADA، متضمناً Introduction to Art and Since of Programming ADA مع Walter J.Savitch، و File Structures with ADA مع Nancy E.Miller.

شكر

وفي نهاية تنظيم الكتاب، نكتب كلمة شكرٍ لكل من ساهم بإعداد ترجمة هذا الكتاب، وتأمين الوسائل الضرورية لذلك.



1

مقدمة Introduction

أزمة البرمجيات

ثقافة «آدا» ADA



إن الكائن البشري، هو الذي بطبيعته صنع واستخدم الوسائل المختلفة. وإن الثورات التي اندلعت باستخدام الوسائل الإجتماعية والعلمية الأكثر تقدماً عن سابقتها، هي التي رسمت تطور النشاطات البشرية المختلفة. وإن إدخال المقالات المكتوبة، والإنتقال إلى تلك المصورة، قد بدَّل البنية الأساسية لحضارتنا. فلقد أنقذت وسائل الطب، مثل المجهر وآلة أشعة X، الكثير من الأرواح. وكثير من وسائل الفنانين، بما في ذلك الغيتار والريشة، قد أغنت أرواحاً كثيرةً أيضاً . وفي كل حالة، خلقت البشرية، أو أتقنت، وسيلةً محددة، لتواجه حاجةً خاصة. والأكثر من ذلك، فقد حسَّنت كل تلك الوسائل فعالية بعض النشاطات، وسمحت بإنجاز أشياء، كانت غير قابلةٍ للتحقيق سابقاً.

وبالمقارنة مع مجال دراسةٍ آخر، تعتبر المعلوماتية علم حديث العهد. له ثوراته، التي تسلسلت بما ندعوه *بأجيال* البنية الصلبة، بدءاً من الصمام المفرَّغ، الترانزيستور، وحالياً الدارات التكاملية. وكما عرض Dijkstra في محاضرته لجائزة Turing، فإن إمكانية هذه الأدوات قد تنامت، لتفوق كثيراً إمكانية السيطرة عليها. فلقد حسَّنت الحواسيب فاعلية بعض الأشياء، وفتحت مجالات تطبيق، كان من غير الممكن قديماً مواجهتها. وبشكلٍ مماثل، فلقد طورنا أدواتٍ برمجية، لمساعدتنا في حل المسائل، والتحكم بآلاتنا. ولكن العديد من هذه الأدوات، مازالت لا تساعدنا بالتغلب على تعقيد الحلول. وبالتالي فإن تطور البرمجيات لم يعد نشاطاً موفِّراً للجهد، لكنه يتطلبه، وبشكلٍ مكثف. وهذا الوضع، هو الذي ندعوه *بأزمة البرمجيات*.

١ ـ ١ ـ أزمة البرمجيات (The Software Crisis):

ظهرت علامات أزمة البرمجيات، على شكل برمجياتٍ لا توافق حاجات المستخدمين، قليلة الوثوقية، كثيرة الكلفة، في غير أوانها، غير مرنة، صعبة الصيانة، وغير قابلة للاستخدام من جديد. وقد امتلأت العقود الأخيرة بمشاريع تخص البرمجيات، والتي أخفقت أو مازالت تتقدم بصعوبة (ظاهرة ندعوها بفساد البرمجي). وحديثاً فقط، بدأنا بفهم تعقيد النظم البرمجية الضخمة، وكيفية السيطرة والتحكم

بتطويرها. فلقد بقينا، ولفترة طويلة، متعلقين بسحر برمجياتنا لتخطي مشاكلنا. ويدعى هذا الارتباط الذي يهدف إلى التخفيف من التحديد البرمجي بـ hacking.

ولحل المشاكل الغامضة لأزمتنا، يجب أن نأخذ الآن طريقة قواعدية للتطوير البرمجي، باستخدام منهجية تصميم مناسبة. ومع ذلك، لا تكفي المنهجية لوحدها لتجابه أزمة البرمجيات، بل يجب أن نملك أيضاً وسيلةً مناسبةً للتعبير، ولتنفيذ تصاميمنا بأي لغة برمجة.

وإن أكثر اللغات الشائعة، FORTRAN وCOBOL، قد تمّ خلقهما مبكراً في تاريخ علم الحاسوب، حتى قبل أن تُفهم المشاكل المتعلقة بتطوير النظم البرمجية الضخمة. وكنتيجةٍ لذلك، فإن هذه اللغات، لا تعكس منهجيات تطوير البرمجيات العصرية. ولقد أجرينا تسويةً بين هذه اللغات والمعالجات الأولية، والإمتدادات، وتحكمات الإدارة، لإجبار هذه اللغات لملاءمة الطرق الأكثر حداثةً. وبمعنى آخر، تقيِّد هذه اللغات طريقنا في دراسة مسألة، وفق خطواتٍ هي بالواقع تسلسلية وإلزامية؛ وندعو هذه المنهجية بـ Von Neuron mind-set.

وأكثر من ذلك، فلقد خُلِقت أدوات هذه اللغات، في زمنٍ كانت فيه المسائل سهلة، إذا ما تمّت مقارنتها بتطبيقات اليوم. ولقد صُممت FORTRAN من أجل التطبيقات العلمية، وCOBOL من أجل التطبيقات التجارية. وحالياً، بقيت هذه اللغات صالحة لمجال المسائل الخاص بكل واحدة. وعلى أي حال، فقد نشأت لاحقاً لتطوير هذه اللغات، عدة مجالات تطبيقاتٍ أضخم، هي مجالات النظم المحمولة، ونظم التحكم بالزمن الحقيقي.

ولم تصمم كل من FORTRAN وCOBOL، ولا معظم لغات البرمجة الأخرى، من أجل مجالات المسائل هذه. وحتى الآن، مازلنا نرى مشاريع تستخدم COBOL من أجل المعالجة بالزمن الحقيقي، أو FORTRAN من أجل تطبيقات متعددة المهام، تتضمن مئاتٍ من آلاف أسطر الترميز. ولا عجبَ إذاً، أن تواجه أزمة برمجيات. فالطرق واللغات التي استخدمناها قديمة، وقد بدأ استخدامها من أجل غاياتٍ غير مخصصة لها.

ويمثل تطوير النظم البرمجية نشاطاً، يتطلب الكثير مـن القـدرة العقليـة، وإكمـال نظامٍ فعال، وقابل للتحقيق في الوقت المناسب، وقابل للصيانة، وقابل للفهم، غالباً مـا يكون مهمةً أكثر صعوبةً واسـتثناءً، خصوصـاً فـي حـال المشـاريع الضخمـة، والبرمجـة بالزمن الحقيقي. والمحترف الذي يخلـق نظمـاً كهـذه، يجـب أن يكـون فـي الوقـت نفسه، علمياً وفناناً.

فمن جهة أولى، يجب أن يكون المبرمج شخصاً علمياً، بمعنـى أنّـه يعمـل على أساسٍ مؤلفٍ من نظرياتٍ واضحةٍ أكيدة، ومجموعة مبادئ مطبقة. فعلى سـبيل المثـال، يمكن للمبرمج أن يستخدم تقنيات البرمجيات، مثل التحليل البنيوي، والتصميم غرضـي التوجه، أو المفاهيم الرياضية لنظرية الأرتال، والتحليـل العـددي. ومـن جهـة أخـرى، يجب أن يكون المبرمج أيضاً شخصاً فناناً، ينحت مركبات نظام من المواد الخام لبنـى المعطيات والخوارزميات، ومن ثمّ يجمع القطع، ليشكل هـذا الكـل. ومـن المحتمـل أن هذه الثنائية، هي التي تجعـل المعلوماتيـة علمـاً أخـاذاً، وتسـبب بالمقابل كثـيراً مـن المشاكل.

طبيعة الأزمة (The Nature of the Crisis):

إن القول بوجود أزمة برمجيات، يمثل صيغـةً مبتذلـة. ففـي العمـق، هـي حالـة واقعة عشنا بها، ولفترة طويلة بعض الشيء. فهي تمثل ميزةً، أكـثر مـن كونهـا أزمـةً مؤقتة. وعلى أي حال، إن الإدراك العام الأول بوجود هـذه الأزمـة، لـم يظهـر إلا فـي المؤتمر العالمي في هندسـة البرمجيـات فـي Garmisch فـي ألمانيـا الغربيـة فـي عـام ١٩٦٨.

وبمعنى آخر، يتمثل جوهر أزمة البرمجيـات، بـأن بنـاء نظمٍ برمجيـة، أصعـب بكثير مما نشعر أو نحس به.

ففي المعلوماتية التقليدية، يمكننا بنـاء نظـمٍ مؤلفـةٍ مـن عـدة مئـات مـن أسـطر الترميز، ونحس بثقةٍ أننا نفهم المشروع المطـروح بشـكلٍ كـامل .وإن تغيـير برمجيـات كهذه ليس صعباً، إذ نستطيع تذكر بنية التصميم، بعد عدة أيامٍ مـن إكمالنـا لنسـختنا

الأولى. وإذا كشف اختبارنا مشاكل ضخمة، يمكننا أن نتجاوزها بسهولة، أو نعاود العمل من الصفر.

وبالمقابل، إن تصميم وتنفيذ نظمٍ مؤلفة من عشرات الآلاف، إن لم يكن ملايين، من أسطر الترميز، يمثل مسألة مختلفة. فالجهد المطلوب لإكمال نظام كهذا، يتعدى القدرات العقلية والفيزيائية لشخص واحد. وعندما نزيد من كادر المشروع لتقاسم المهمة، تتضاعف المشكلة، إذ أننا ندخل بذلك مشاكل إضافية، تتعلق بموضوع التواصل بين هذه الأطراف. وبالتالي، فإن تغيير نظامٍ كهذا يكون صعباً، لأن البنية المتكاملة للمشروع، لم تتعلق بشخصٍ واحد. وإذا كشف الإختبار مشاكل ضخمة، فعادةً لا نملك وسائل فذة (من الزمن أو المال)، لإعادة تصميم النظام بكامله مجدداً.

وكل مبرمجٍ يعمل في نظم برمجيةٍ كثيفةٍ وضخمة، التمس سعادة ابتكار وإيجاد حلٍ لمسائله، ويزداد إحباطه كلما تنامت مشاكله. وإذا طلبت من ذلك المبرمج أن يخبرك ما هو سبب المشكلة الأساسية، تتلقى أجوبةً مختلفة، مثل، "لتلك الوحدة تأثيرات جانبية غريبة"، أو "إن واجهة التخاطب، لم تكن معرَّفة بشكلٍ حسنٍ وكاف"، مثلما سنرى، وهذا ما هو بالحقيقة، إلا أعراض لمشاكل غامضة. وبشكلٍ عام، تقود هذه العلامات والدلائل جميعها، لنظمٍ برمجيةٍ بطيئة، ومكلفة، وغير قابلة للتحقيق، وغالباً، غير ملائمةٍ أو متطابقةٍ مع توصيف النظم. وهذه المشاكل، هي العوامل الأكثر وضوحا في أزمة البرمجيات، وبشكل بديهي، ندرك وجود مشاكل برمجية، لكنه من الصعب إدراك تأثيراتها الكاملة. ومثلما قال David Fisher، "بالرغم من وجود الكثير من العلامات والدلائل المعروفة، فإن المشاكل الغامضة ما زالت غير محددة تماماً، وهناك قليل من القياسات الكمّية المفيدة، التي تسمح بتقدير سواءٍ أهمية المشاكل المكتشفة، أو فعالية الحلول المطروحة". كما ذكر بعض علامات أزمة البرمجيات مثل:

- ***الإستجابة***: غالباً لا تلبي نظم الحاسوب الأساسية حاجة المستخدم.
- ***الوثوقية***: غالباً ما يتعطل البرنامج.
- ***الكلفة***: نادراً ما يتم التنبؤ عن تكلفة البرنامج، وغالباً ما تكون مفرطة.

- *قابلية التغيير*: صيانة البرنامج معقدة، ومكلفة، وعرضة للخطأ.
- *تجاوز الزمن*: غالباً ما يتأخر البرنامج، حيث لا ينتهي ضمن الزمن المسموح به.
- قابلية *النقل*: نادراً ما يعمل البرنامج على عدة نظم، حتى عندما تُطلب الوظائف نفسها.
- *الفاعلية*: لا تستخدم جهود التطوير البرمجي استخداما أمثلياً، الموارد الجاهزة (زمن المعالجة وسعة الذاكرة).

وبالطبع لا تمثل التكلفة، المؤشر الوحيد لمشاكل البرمجي؛ بينما تبقى مسألة نوعية البرنامج هي الأساس. وهناك العديد من الأمثلة عن مشاريع برمجيةٍ ضخمة، والتي لم تراع المخطط الزمني للمشروع، وتجاوزت التكلفة المقدرة، ولم تنجز سوى جزءٍ من دفتر الشروط والمتطلبات الفنية.

الأسباب الرئيسية للأزمة (Underlying Causes of the Crisis):

حتى الآن، ذكرنا فقط علامات أزمة البرمجيات. وللتغلب على المشكلة الفعلية، يجب علينا أولاً، أن نفهم الأسباب الأساسية لوجودها. إنّ المطور البرمجي، يسلك غالباً سلوك الفنان. ومع ذلك، عندما نزجه في بيئةٍ هندسية، غالباً ما تكون النتيجة غير جيدة. وهذا لا يعني بأنّ البرمجة مجردة من الإبداع. فعلى العكس، هي ومضة الإبداع التي تدعنا نرى الحلول الفريدة من نوعها. وتأتي المشكلة، عندما نطبق فناً بشكلٍ غير مدروس. فبالنسبة للموسيقي، ينتج تنافر في النغمات؛ أما بالنسبة للمبرمج، فينتج نظام برمجي مكلف، لا يمكن تحقيقه، ولا يمكن صيانته.

وقد Devlin عدة أسباب ضعفٍ لهذه الثقة، لفنٍ برمجي غير مدروس، منها:

- إخفاقٌ في تنظيم وفهم دورة الحياة، التي تؤدي لتطور البرنامج.
- نقصٌ في عدد الأشخاص المؤهلة في هندسة البرمجيات.
- إن أُسس Von Neumann لمعظم آلاتنا، لا تشجع استخدام أساليب البرمجة العصرية.
- إتجاه التنظيمات بالإعتصام باستخدام لغات برمجة قديمة وعملية.

ولقد أكد على النقطة الأخيرة بإضافة "ولا يوجد مسؤول يقبل بالجيل الأول من الحاسبات، والذي يعتمد على الصمام المفرغ. وبالمقابل، يوجد قلة منهم يريدون التخلي عن لغات البرمجة من الجيل الأول، مثل الـ FORTRAN".

فنحن نواجه مشاكل تطبيقية معقدة، لكن الطرق، واللغات، والأدوات الموجودة حالياً تخفق بمساعدتنا لإيجاد حلولنا. ولا نستطيع أن نأمل باختصار تعقيد مسائلنا؛ وكلما أصبحت أدواتنا أفضل، كلما نكشف باستمرار، مجال مسألةٍ ذات تعقيد متزايد.

مقاومة الأزمة (Combating the Crisis):

قد يبدو وجود حدود إنسانية أساسية، تعوقنا عن إيجاد حلولٍ لمسائل برمجياتٍ معقدة. وفي الواقع، ليست هذه الحالة. فعلى سبيل المثال، من الصعب على شخص، حفر ثقبٍ صغيرٍ بدون شيء؛ ومن المستحيل لأي شخصٍ، أن يلتقط المحار من قناة بنما دون شيء، على الرغم من البناء الأكيد للقناة: وأنّ حل المسألة، كان باستخدام الوسائل القوية، التي وسّعت مقدرة البنائيين. وينطبق هذا الشيء على مطوري البرمجيات؛ حيث يجب أن نطبق طرقاً، ولغات برمجة، وأدواتٍ تساعدنا، ولا تعقد حلولنا البرمجية.

ويوجد العديد من الأدوات البرمجية هذه ، مثل، تقنيات البرمجة البنيوية، ومخططات تدفق المعطيات، وطرق التصميم غرضية التوجه، ومحيطات التطوير المتكاملة. ومثلما سنرى بالتفصيل في الفصول القادمة، تعتمد هذه الأدوات، على مجموعةٍ من المفاهيم البرمجية الأساسية. وأكثر من ذلك، فقد أشارت الدراسات، إلى أنه في غياب طرق البرمجة العصرية، تكون الإنتاجية ثابتةً نسبياً (حول عشرة أسطر ترميزٍ منقحةٍ باليوم، من قبل شخصٍ واحد)، وذلك بصرف النظر عن مستوى لغة البرمجة المستخدمة. ويقودنا هذا لاختيار لغات برمجةٍ عالية المستوى لتنفيذ حلولنا، حيث أنّ كل تعليمة فيها أكثر فعاليةً من تعليمةٍ مكتوبةٍ بلغة المجمع. ومع ذلك ، يجب أن نكون حريصين باختيار اللغة، لأن كل لغة ستؤثر وبشدة. على كيفية تصميم حلولنا بشكلٍ نهائي. وهذا ما أشرنا إليه سابقاً، بأن هناك مجموعة أفكار، تقيدنا بالتفكير في طرقٍ محدودة.

والحل النهائي للمسألة، ضمن أزمة البرمجيات – المسمى، حدودنا الإنسـانية – يكمن في تطبيق الطرق البرمجية العصرية، المدعومة بلغة برمجةٍ عالية المستوى، والتي تشجع وتدعم هذه المفاهيم، في محيطات تطويرٍ ملائمة. وفي المقطـع التـالي، سنوجز تطوير إحدى لغات البرمجة عالية المستوى.

١ ـ ٢ ـ ثقافة ADA (The ADA Culture):

كرد فعلٍ على أزمة البرمجيات، فقد موّلت وزارة الدفاع الأمريكيـة عمليـة تطويـر لغة برمجةٍ قويةٍ جـداً، هـي لغـة ADA. وبشـكل مختلـفٍ عـن معظم لغـات البرمجـة الأخرى، فقد تمّ تصميم ADA لمجال مسـألة معيـن، هـو مجـال النظـم التـي تتطلـب برمجيات قوية، والتي تحتاج إلى لغة ذات مجموعة متطلباتٍ محددة. ولم تصمم ADA من قبل لجنة، ولكـن مـن قبـل فريـق تصميـمٍ صغـير. ومـن ثـم تـمّ تنقيحهـا مـن قبـل مستثمرين، وخبراء في هذا المجال .

وتمثل ADA "تقدماً هائلاً في تقنيات البرمجة، إذ تضم أفضل الأفكار، وبطريقـة متماسكة، ومصممة لتواجه الإحتياجات الحقيقية للمبرمج العملي". ولم تكن ADA معفاة من النقد، على أي حال. حيث أن معظم المناقشات صرحـت، بأنهـا معقـدة، وليسـت عملية. ولقد رفضنا ذلك بقوة. إذ تأسست ADA على مجموعة صغيرة من مفـاهيمٍ سـهلة الفهـم، مثـل تجريـد المعطيـات، وإخفـاء المعلومـات، والتنويـع القـوي. وبسـبب هـذه الأسس، يمكن لأي شخص استخدام ADA بشكلٍ فعال وسريع، بتعلّم مجموعةٍ جزئيـةٍ من ميزاتها، وبالتدريج، تعلم اللغة الطبيعية.

وبمعنى آخر، تضم ADA العديد من مفاهيم هندسة البرمجيات العصرية، ووسيلةً ممتازةً للتعبير عن الحلـول البرمجيـة. ولا تشـجع ADA فقـط علـى الاسـتخدام العملـي والسهل للتصميم والبرمجة، مثلما سـنرى فـي فصـول لاحقـة، بـل يمكنهـا بالحقيقـة، تدعيم تلك الإجراءات العمليـة السـهلة. وبشـكلٍ مشـابهٍ تماماً لبقيـة الوسـائل البشـرية الثوريـة، فـإن ADA، تسـاعد فـي الوصـول إلـى طـرق جديـدة، وغالبـاً أكـثر فاعليـة. وبالاضافة إلى تطوير لغة ADA، فقد طورت وزارة الدفاع الأمريكيـة، مـا يسـمى بدفـتر

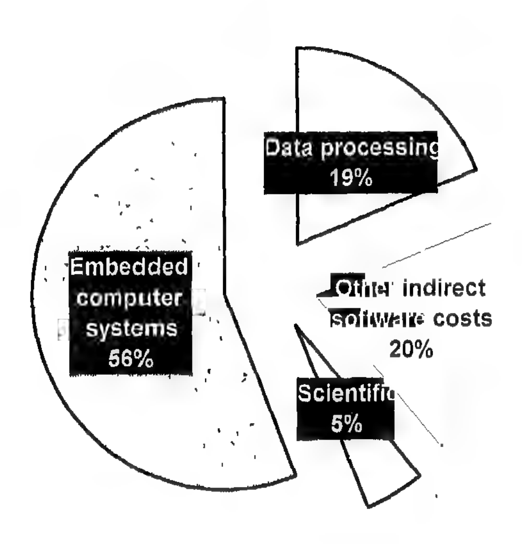
الشروط الفنيــة لبيئــة البرمجـة بلغـة ADA وسـمّته: ADA Programming Support Environment.

وإن لغة ADA والبيئة APSE المقترنة بها، وطريقة المحاكمـة الخاصـة بـ ADA، تسمى ما ندعوه "ثقافة ADA"، حيث تهدف هذه البيئة APSE، إلـى تقديـم جميـع وسائل التطوير بلغة ADA.

مرحلة التحليل (Analysis Phase):

بشكل نظري، وبالرغم من أنه يمكن التعبير عن أي حسابٍ بلغـةٍ محـددة، فإنـه يمكن التعبير بأية لغة أخرى. والحلول الناتجـة، غالبـاً، تتغير بشـكل واضح. وتكـون بعض الحلول أكثر فهماً من البقية. وكما سنرى فـي الفصـل التـالي، فـإن لغـة البرمجـة تجسد طريقة تفكيرنا حول حل المسائل. ونحن نحتاج للغة برمجة، تقودنا لنظمٍ تطـابق مباشرةً مجال المسألة – لغة تساعدنا للتحكم بتعقيـد الحلـول البرمجيـة. وADA تمثـل هذه اللغة. وبالرغم من أن ADA ليست لغة البرمجة النهائية، لكنها هي المناسبة لخلق نظم ضخمة، لنظمٍ يمكن تحقيقها، ويمكن صيانتها. وبالطبع، لم تخلق ADA من فراغ، لكنها حصيلة نتاج معالجة إنسانية متطورة. وسوف نلقـي الضـوء علـى هـذه المعالجـة، ونقدم عرضاً لهذا الجهد. إن مشاكل التطوير البرمجي لم تكن مفهومة بعـد، بـالرغم مـن أن كـل مشـاكل آثـار الإدارة السيئة، والتـأخير فـي التسـليم، والبرمجيـات غـير الموثوقة، وتجاوز التكلفة، قد أُدركـت تمامـاً. ففـي عـام ١٩٧٣، شـملت البرمجيـات لوزارة الدفاع الأمريكي ما يقـارب ٤٦٪ (أكـثر مـن ٣ بليـون دولار)، مـن أصـل الكلفـة الكلية في مجال المعلوماتية، والتي بلغت ٧,٥ بليون دولار. وأكثر من ذلك، فـي أوائـل عام ١٩٧٠، أدى اهتمام وزارة الدفاع الأمريكية، إلى زيادة تكلفـة البرمجيـات لمعظـم نظم الدفاع. وقبل ذلك الزمن، تجاوزت تكلفة البنية الصلبة، كثيراً، تكلفـة البرمجيـات لتطوير هذه النظم. وإن مشاكل تطوير البرمجيات الضخمة، وكما هو موضح فـي الشـكل التالي، فإن ٥٦٪ من تكلفة البرمجيات، قد تمّ إنفاقها على قطاع النظم المحمولة، بينما معالجـة المعطيـات (باسـتخدام COBOL بشـكل أساسـي) شـملت ١٩٪، والتطبيقـات

العلمية (بشكل عام مكتوبة بـ «فورتران» FORTRAN)، شملت ٥٪. والباقي، والذي يقدر بـ ٢٠٪، تضمن تكاليف برمجية أخرى، غير مباشرة.



الشكل ١ ـ ١. الكلفة البرمجية المتوقعة لوزارة الدفاع، في عام ١٩٧٣.

التقدير الكلي لوزارة الدفاع الأمريكية، لتكلفة البرمجيات في عام ١٩٧٣:

من عام ١٩٦٨ لعام ١٩٧٣، سجلت وزارة الدفاع الأمريكية زيادة مقدارها ٥١٪ من التكلفة المباشرة لنظمها الحاسوبية، بالرغم من تناقص تكاليف البنى الصلبة (التجهيزات) بشكل مشهود. وبالحقيقة، لا تشير هذه التكاليف إلا لجزء فقط من المشكلة، التي واجهتها وزارة الدفاع الأمريكية. وسنكون قادرين على قبول القول "حسناً، هذا مكلف، لكن متطلباتنا تبرر تكلفة منتجٍ بنوعيةٍ عالية"، على أي حال. نادراً ما نحصل على برمجياتٍ نوعية.

وبالإضافة لذلك، كان هنالك عدد من العيوب الأساسية، والتي شكلت حاجزاً لحل المسألة الأساسية، وهي:

- وجود لغات برمجةٍ متعددة.

- إستخدام لغات برمجةٍ بشكلٍ غير مناسب لتطبيقاتها.
- إستخدام لغات البرمجة، التي لا تدعم المبادئ العصرية لهندسة البرمجيات.
- النقص في محيطات تطوير برمجياتٍ نافعة.

ويوجد في وقتنا الحالي على الأقل، ٤٥٠ لغة برمجة، ذات أهداف، عامة لنظم وزارة الدفاع الأمريكية، بالرغم من أن، (حسب مصدر المعلومات)، العدد الحقيقي يتغير بين ٥٠٠ و١٥٠٠ لغة برمجة عالية المستوى، مختلفة ولغات المجمع. وبما أنّ وزارة الدفاع الأمريكية لم تكن تملك نقطة تحكمٍ وحيدةٍ لكل لغة، فكل إدارة مشروعٍ كانت حرةً فعلياً، بخلق لغتها الخاصة بها، أو أن تستخدم لغة فرعية، من لغةٍ برمجية غير متوافقة. وينتج عن هذا بعثرة لجهود التدريب. فعلياً، لا يوجد تبادل تقنيات بين المشاريع، وبالتالي، بعثرة عامة للموارد.

وهنالك العديد من الحالات، تكون المشاريع فيها مرتبطة ببائعين خاصين، أو بتقنياتٍ قديمة؛ ولهذا، كنا نرى غالباً، مشاريع تستخدم COBOL من أجل تطبيقات الزمن الحقيقي، أو لغات المجمع للنظم التجارية. وإن العديد من هذه اللغات، والتي تمّ تطبيقها، لا تدعم مبادئ هندسة البرمجيات العصرية. ففي بداية ١٩٧٠، حتى البرمجة البنيوية لم تكن مفهومة بشكلٍ جيد، ولا مقبولة بشكلٍ حسن.

إن هذا التكاثر في اللغات أيضاً، يغير مجرى الموارد الشخصية، ويحول دون خلق أدواتٍ برمجيةٍ فعالة . ففي الواقع، لم يكن يملك المشروع سوى أدواتٍ بدائية، مؤلفة على الأكثر، من المترجم، والرابط (Linker)، المحمل. وعلى العكس، فإن محررات القواعد المباشرة، وأدوات إدارة التشكيلات (configuration)، والترجمة المتزايدة، لم تكن معروفة لأكثر المشاريع البرمجية؛ وكان المبرمجون ببساطة، مشغولين بالتصارع مع برمجياتهم، ليعالجوا ما كانوا يدركوه، كسحرٍ نظري أو أكاديمي.

وفي كانون الثاني من ١٩٧٥، فإن Malcolm Currie، مدير هندسة وبحوث الدفاع (DDR&E)، أدرك بأنّه ستكون هنالك فوائد عديدة باستخدام لغة برمجة وحيدة، مشتركة، وعالية المستوى. وهكذا، فقد عين قطاعاً مشتركاً لمجموعة عمل، من أجل لغةٍ عالية المستوى (HOLWG) . ولقد ترأس مجموعة العمل للغة عالية المستوى

Lt.Col. William Whitaker مـن القوى الجويـة الأمريكيـة (USAF). وتضمنـت هـذه المجموعة ممثلاً عن كل القوات، بالإضافة لمكاتب أخرى من وزارة الدفاع، وكانت على اتصالٍ متبادلٍ مع المملكة المتحدة، وألمانيا الغربية وفرنسا. وباختصار، لقد كلفت هـذه المجموعة بما يلي:

- تعيين دفتر الشروط الفنية للغات عالية المستوى، لوزارة الدفاع الأمريكية.
- تقييم اللغات الموجودة، بالنسبة لدفتر الشروط هذا.
- تبني أو تنفيذ مجموعة دنيا من لغات البرمجة.

مرحلة تعريف دفتر الشروط الفنية (Requirements Definition Phase):

في نيسان من عام ١٩٧٥، قامت مجموعة العمل HOLWG بوضع دفـتر الشـروط الفنيـة STRAWMAN لأقسـام الجيـش، ولمكـاتب فيدراليـة أخـرى، وللصناعـة، وللجامعة. وقد تمّ التماس التعليقات أيضاً، من خبراء مختارين في المجموعـة الأوروبيـة للمعلوماتية، مثل Dijkstra و Hoare .

وبعد تدقيق ومراجعة دفتر الشروط STRAWMAN، تمّت كتابة وتوزيع الوثـائق المسماة WOODENMAN في آب من عام ١٩٧٥، مرة ثانية، ولأوسع جمهـورٍ ممكـن. وأدت الإستجابات الرسمية لهذه المراجعـات، إلـى وضـع دفـتر شـروطٍ متكـاملٍ دعـي بـ TINMAN، والذي وزّع في كانون الثاني مـن عـام ١٩٧٦، وكـان يمثـل المواصفـات المرغوبة للغة عالية المستوى لوزارة الدفاع الأمريكية.

وتمّت الكتابة الأولية لكل واحدة من هذه الوثائق، من قِبـل David Fisher، فـي معهد تحليل الدفاع بمساعدة P.R. Wetherall.

وفـي عـام ١٩٧٦، نشـرت التعليمـات DOD 5000.29 بعنـوان "إدارة المـوارد المعلوماتية، في أنظمة الدفاع الأساسية".

وبالتوازي مع ذلك، بدأت مجموعة العمل HOLWG بتقييم اللغات، التي تعتمـد مواصفات TINMAN . وقد استمر هذا التقييم طيلة عام ١٩٧٦، من قِبل ستة متعهدين أساسيين، وعددٍ كبيرٍ من الأشخاص. وفي نفس الوقت، طورت HOLWG لائحةً مؤقتةً

عن لغات عالية المستوى موافق عليهـا، وحُوِّلت إلـى وثيقـةٍ رسـميةٍ مـن إدارة DOD تحت رمز 5000.31، ونشرت في عام ١٩٧٦. وكل قطاعٍ يشتمل علـى لغـاتٍ أساسـية، وقد تم اختيار بعض اللغات الموافقة لأي قطاع فيDOD. وقـد تضمنـت لائحـة اللغـات التي تمت الموافقة عليها، ما يلي :

وزارة الدفاع الأمريكية	FORTRAN
	COBOL
الجيش (المشاة)	TACPOL
القوى البحرية	CMS-2
	SPL/1
القوى الجوية	JOVIAL J3
	JOVIAL J73

وفي كانون الثاني من عام ١٩٧٧، اكتمل تقييم لغات البرمجة الموجودة، بالنسبة لمواصفات TINMAN ، وبالإجماع، قدمت التقييمات أكثر من ٢٨٠٠ صفحـة تعليـق. ولقد فحص فريق المراجعة بشكل رسمي، أكثر من ٢٣ لغة مختلفة :

FORTRAN, COBOL, PL/1, HAL/S, TACPOL, CMS-2, CS-4, SPL/1, JOVIAL J3, JOVIAL J73, ALGOL60, ALGOL68, CORAL66, Pascal, SIMULA67, LIS, LTR, RTL/2, EUCLIDL K MORAL, & EL/1.

واستنتج هذا التقرير ما يلي :

- لا توجد أي لغـة مـن بيـن اللغـات المذكـورة أعـلاه، مناسـبة لاسـتخدامها كلغـةٍ مشتركة عالية المستوى، للنظم المحمولة لوزارة الدفاع الأمريكية، بحيث تكون:
- لغة وحيدة مرغوبة.
- لغة جديدة، تواجه المتطلبات التي تعتبر ملائمة.
- وبحيث يمكن تطوير هذه اللغة الجديدة اعتباراً من أساس مناسب.

وبالرغم من أن المقيّمين شعروا بأنّ جميـع اللغـات التـي تمّـت مراجعتهـا غـير مطابقة، فقد زكّوا أخيراً PL/1، و ALGOL، و Pascal،كلغاتٍ أساسيةٍ مناسبة.

ولغاية كانون الثاني من عـام ١٩٧٧، حتى اكتمـل تقييم دفـتر الشـروط الفنيـة TINMAN، الذي استغرق العام كله، وحيث تم نشر وثائق IRONMAN، التي دعمت تعليقات المراجعين. ثم تمّ إنجاز تحليلين اقتصاديين مستقلين اعتباراً من كـانون الثـاني من عام ١٩٧٧، وحتى تشرين الثاني من نفس العـام، لتحديـد مـا إذا كـان مـن العملـي تطوير لغةٍ جديدة، معتمدة على دفتر شروط TINMAN . وكانت نتـائج كـلا التحليليـن إيجابيةً بشكل جيد، وأشارت بأنّ تبني لغة برمجةٍ مشتركةٍ وعالية المسـتوى، سـيؤدي إلى توفير مئات الملايين من الدولارات لوزارة الدفاع الأمريكية، في كل عام.

مرحلة التصميم (Design Phase):

بالنسبة إلى وزارة الدفاع الأمريكية، فإن تصميم هذه اللغة الجديدة، يجـب أن يكـون من نوعيةٍ عالية، خصوصاً أنه سيصبح فيما بعـد، معيـاراً مشـتركاً . ولذلـك، فقد أدركـت DOD، بأنّ فوائد ضخمة ستنتج من لغةٍ تمّ قبولها بشكلٍ حسن، من خـارج جماعـة الدفـاع العسكرية. ومن أجل هذه الأسباب، تمّ طرح مسابقة تصميمٍ عالميٍ منافس.

وفي تموز من عام ١٩٧٧، تمّ اختيار أربع متعهدين. وبنفـس الوقـت، تـمّ توزيـع نسخةٍ منقحةٍ من دفتر شروط «آيرونمان» IRONMAN . وإن تصميمات اللغات الأربـع التي تمّ قبولها من أجل المرحلة الأولى، تضمنت إقتراحاتٍ من:

- SofTech (Blue).
- SRI International (Yellow) .
- Intermetrics (Red).
- Honeywell/Honeywell Bull (Green).

ولقد تمّ ترميز المقترحات بالألوان، إلى حد أنّ المراجعين لـن يعرفـوا مصادرهـا. ومـن المهـم ملاحظـة أنّ جميـع هـؤلاء المتعهديـن، اسـتخدموا Pascal كلغـةٍ أساسـيةٍ لتصميماتهم للغة الجديدة.

وقد انتهت المرحلة الأولى من التقييم، فـي شـباط ١٩٧٨. ومنـذ شـباط مـن عـام ١٩٧٨، وحتى آذار من نفس العام، تمّ تقييم التصاميم على المستوى العالمي، بواسـطة

٤٠٠ متطوع مقسمين في ٨٠ فريقاً . واعتماداً على هذه التقييمات، تمَّ اختيار تصميمين، الأول المقترح من قِبل Intermetrics (Red)، والثاني المقترح من قِبل Honeywell/Honeywell Bull (Green). ليتم إجراء دراسة عليها أكثر عمقاً. وهنا بدأت المرحلة الثانية .

وفي بداية ١٩٧٧، أدرك Whitaker، بأنّ لغة برمجةٍ ليست كافية لوحدها لتأكيد التحسينات المرغوبة في التطوير البرمجي، بل تحتاج أن تُربط اللغة مع أدوات نوعية. وفي بداية عام ١٩٧٨، نشرت HOLWG وثيقة تُدعى SANDMAN، والتي طُرحت فيها بعض التساؤلات التقنية والإدارية، ذات العلاقة بمحيطات اللغة؛ وقد تمَّ أخيراً تنقيح SANDMAN لتشكيل PEBBLEMAN. وخلال نفس الشهر، نشرت HOLWG النسخة النهائية تحت إسم STEELMAN.

وفي تشرين الثاني من عام ١٩٧٨، نظّم مكتب بحوث الدفاع المتقدمة، لقاءً بين مصممي لغتي Red و Green ، إذ سُمح للمشاركين بطرح أسئلةٍ نوعيةٍ عن التقنية للغة الجديدة. وانتهت المرحلة الثانية في آذار ١٩٧٩.

وفي ربيع عام ١٩٧٩، اقترح Jack Cooper من قيادة الأسلحة البحرية، الإسم الكامل لهذه اللغة الجديدة: ADA تكريماً لـ Augusta ADA Byron أميرة Lovelace وابنة الشاعر Lord Byron .

وكانت (١٨١٥ ـ ١٨٥١) ADA Lovelace عالمة رياضيات، عملت مع Charles Babbage على محركاته التفاضلية والتحليلية. ولقد عُرفت بملاحظاتها المبكرة على القوة الكامنة للحاسوب . وبشكل خاص، فقد اقترحت ADA كيفية إمكانية برمجة آلات Babbage، من خلال مهنة الـ Jacquard . ومن أجل هذا العمل، فقد اعتبرت أول مبرمجٍ في العالم. وفي تبادلٍ رسميٍ للرسائل، بين وكيل سكرتارية الدفاع، وورثة Lovelace، فإنّ Earl of Lytton، منح رخصة تسمح باستخدام هذا الإسم.

وفي أيار من عام ١٩٧٩، أعلنت HOLWG اللغة الخضراء، وهي أوروبية، كلغةٍ ربحت منافسة التصميم. وكان الفرنسي Jean Ichbiah المؤلف الأول لهذه اللغة. ومن

الأعضاء الآخرين في فريق التصميم كان J.Abrial, & O.Roubine, & J.Heliard، من فرنسا أيضاً؛ و P.N.Hilfinger & H.F.Ledard من الولايات المتحدة؛ M.Woodger، و B.A.Wichmann، و J.G.P.Barnes، و R.Firth من المملكة المتحدة؛ و Krieg-Bruckner من ألمانيا الغربية. كما أن Reference Manual for the ADA Programming Language قد عرَّف عدة أشخاص آخرين، قدّموا مساعداتٍ معبّرة لتصميم اللغة، منهم، E.Morel & G.Ferran من فرنسا؛ J.B.Goodenough, M.W.Davis, L.Maclaren, I.R.Nassi, S.A.Schuman, & S.L.Vestal من الولايات المتحدة؛ وI.C.Pyle من المملكة المتحدة.

وفي ١٢ كانون الأول من عام ١٩٨٠، خلال مؤتمر الـ ACM الأول لـ ADA، المعقود في Boston، تمّت الموافقة على خلق مكتب برنامجٍ مشترك لـ ADA، لإدارة جميع النشاطات المتعلقة بـ ADA . ولقد تمّ حل HOLWG بشكل رسمي، وسُمي Lt.Col.Larry مديراً لـ AJPO . وفي نفس ذلك اليوم، تمّت الموافقة على تأسيس MIL-STD 1815 كمعيار لـ ADA مصدق عليه من وزارة الدفاع الأمريكية. وسنضيف بأنّ اختيار العدد ١٨١٥ لم يكن صدفة، إذ أنّ ١٢ كانون الأول ١٨١٥ كان تاريخ إعلان ولادة Augusta ADA Lovelace .

وفي ١٧ شباط من عام ١٩٨٣، تمّت الموافقة على ADA بأنها معيار لـ ANSI . وفي نفس العام، وُجدت أول مفسرات ومترجمات صالحات للعمل. وفي آذار من عام ١٩٨٧، قبلت ISO (منظمة المعايير العالمية) معايير ADA الأمريكية والفرنسية.

تأثير ADA على هندسة البرمجيات

(The Impact of ADA on Software Engineering):

لقد لاحظ عالم النفس الشهير Benjamin Whorf، بأن اللغات " يمكن أن تملك تأثيراً هاماً على المعالجات الفكرية، حتى لو لم تحدد تكامل هذه المعالجات". وهكذا، نتوقع نفس الشيء من ثقافة ADA. ولقد ساعدت ADA بكسر مجموعة أفكار Von Neumann، وتركتنا نبحث في حلولنا، بدلالة مجال المسألة الحقيقي. وكنتيجةٍ

لذلك، أصبحت الحلول مقروءةً بشكل أفضل، وموثوقةً، ويمكن صيانتها. وفي الواقع، تسمح لنا ADA بالبحث في مجموعة من المسائل كلها جديدة، إذ كانت إدارة حلولها قديما معقدة.

ولقد كتب Fisher، " ليست لغات البرمجة سبباً، ولا حلاً للمسائل البرمجية. ولكن بسبب الدور المركزي الذي تلعبه في جميع نشاطات علم الحاسوب، يمكن للغات البرمجة أيضاً، أن تزيد من تعقيد المسألة، بدلاً من أن تبسّط حلولها". وفي حالة ADA، يمكن أن نتوقع أن يكون تأثرها في تبسيط الحلول. وبالرغم من أن ADA لا يمكنها بمفردها أن تحل أزمة البرمجيات، لكن عند ضمها مع طرقٍ وأدواتٍ برمجيةٍ كاملة، يمكن أن تقدم وسيلةً قادرةً على خلق نظمنا البرمجية.

لقد وُضعت ADA بالإستثمار، منذ أكثر من ثمان سنوات. وإننا نوافق بأننا حالياً نحاول تكميم تأثيرها. فقد كان هنالك العديد من التقارير المنشورة عن زيادة الإنتاجية، كنتيجةٍ لاستخدام ADA . وقد قارنت معظم هذه الدراسات مشاريع ADA الأساسية، مع مشاريع ذات متطلباتٍ شبيهةٍ جداً في الـ FORTRAN أو C . وقد عبّرت جميع التقارير عن زيادة الإنتاجية. وبالإضافة لذلك، فقد أظهرت دراسات عديدة، بأنّ الإنتاجية استمرت بالزيادة عند العمل مع فريق برمجة بلغة ADA. وقد تغيّرت زيادة الإنتاجية من ٣٠٪ وحتى ١٠٠٪. وإن جزءاً ضخماً من هذه الأرباح، يمكن أن يعزى إلى انخفاض في الأخطاء البرمجية، وإلى زيادةٍ في إعادة استخدام البرنامج. وقد كتب Reifer، في تقريره " حول تحليل القياسات المأخوذ من عشرات المشاريع، " بأنّ مشاريع لغة ADA تولد أخطاء أقل بـ ٣٠٪ من الأخطاء الكلية، و٢٠٪ من الأخطاء الحرجة بعد التسويق. ولقد قارن Doscher نظاماً تمّ تنفيذه بلغة C، مع نظامٍ تمّ تنفيذه بلغة ADA ووجد أن أخطاء أقل بـ ٩٦٪، بعد تسويق المنتج المنفذ بـ ADA.

وقد قارنت دراسة من NASA بين FORTRAN مع ADA . ثلاث مشاريع متتالية في ADA، نفذها نفس الفريق، تمّ فحصها. فوجد بأنّ أول نظام ADA ولّد

أخطاء أقل بـ ٥٠٪ من نظام فورتران مشابه قبل التسويق. وفي المشروع الثالث، قلص فريق ADA أخطاء التسويق بأكثر من ٦٦٪.

وقد قارن Hines مشروعاً تمّ تنفيذه بـ C، مع مشروع تمّ تنفيذه بـ ADA. وكان مشروع ADA، الأول المتعهد من قبل فريق العمل. ووجد بأنّ مشروع C، يقدم ٠٪ من فرصة إعادة استخدام البرمجيات، بينما مشروع ADA، فقد أعاد استخدام ١٠٪ من برمجياته البدائية.

وجد Reifer أيضاً، بأنّ فريق مشروعهم الثالث بـ ADA، كان قادراً على إعادة استخدام ٢٠٪ من برمجياته.





2

هندسة البرمجيات
Software Engineering

أهداف هندسة البرمجيات.

مبادئ هندسة البرمجيات.

طرق تطوير البرمجيات.

اللغات، وتطوير البرمجيات.



يتمثل السبب الأساسي لأزمة هندسة البرمجيات، بأنّ النظم الضخمة في البرمجيات القوية، أصبحت ذات تعقيدٍ يفوق مقدرتنا. وأكثر من ذلك، لا يمكننا أن نتوقع تخفيض ذلك التعقيد، لأنه كلما حسّنا أدواتنا، وزدنا خبرةً في تصميم هكذا نظم، نكون بالحقيقة، قد فتحنا باب مسائل أكثر تعقيداً. وكحل لهذه الأزمة، يجب عندها أن نطبق فناً مدروساً، مستخدمين أدواتٍ تساعدنا في إدارة هذا التعقيد. وبمعنى أوسع، ندعو هذه الدراسة *بهندسة البرمجيات*. وعندما نفكر بهندسة البرمجيات، غالباً ما يجب أن يتبادر للذهن رؤية الترميز البنيوي، PDL (لغة تصميم البرنامج)، رسم الـ HIPO (دخل ـ معالجة ـ خرج الهرمية)، ومخطط تدفق المعطيات. وفي الحقيقة، تمثل هذه الأمور، فقط، قليلاً من براعاتٍ متعددة في هندسة البرمجيات. وببساطة محددة، فإن الغاية من هندسة البرمجيات تتمثل بتقديم طريقةٍ متماسكةٍ لخلق نظمٍ برمجية، آخذة بعين الإعتبار دورة الحياة. ويمكن أن نقول، بأننا نبحث، بعملنا، تبديل بعض السحر، بمبادئ هندسية عملية وجيدة . وفي هذا الفصل، سنفحص، وبطريقةٍ أكثر تفصيلاً، أهداف هندسة البرمجيات. وبعد ذلك، سندرس بعض المبادئ البسيطة، التي تساعدنا بالوصول لهذه الأهداف. وبعد ذلك، سنقدم عدة طرق لتطوير هندسة البرمجيات، والتي تطبق هذه المبادئ. وأخيراً، وبعد معرفة أنّ لغاتٍ البرمجة تمثل وسائل تسمح لنا بالتعبير وتنفيذ تصاميمنا، سنفحص أجيال اللغات، وندرس كيف تدعم أو لا تدعم كل واحدة، المبادئ الأساسية لهندسة البرمجيات. ومثلما بدأنا بفحص التفاصيل التقنية لـ ADA في هذا الفصل، سنرى بأن تجسيد مبادئ تطوير البرمجيات هذه، هو في الواقع وسيلة ممتازة لخلق نظم برمجية.

٢ ـ ١ ـ أهداف هندسة البرمجيات

(Goals of Software Engineering):

من البديهي بأنّ أحد الأهداف الرئيسية لتطوير برنامجٍ ما، يتمثل بتحقيق الحل الناتج عن متطلبات خاصة. وعلى أي حال، فمن النادر وجود متطلباتٍ كاملةٍ أو متماسكة، خاصةً من أجل النظم الضخمة جداً. وغالباً، لا يفهم المستخدِم (والمنفذ)

المسألة بشكل كامل، وبالتالي، يستنبط المتطلبات عادةً خلال تطوير النظام. وأكثر من ذلك، تتضاعف المسألة عندما يكون لدينا على التوازي، تطويرٌ للبرمجيات، وتطويرٌ للبنية الصلبة، وغالباً، هذه هي الحالة في النظم المحمولة. وأخيراً، يجب أن نقبل حقيقة تغيرات المتطلبات، خلال دورة حياة نظمنا البرمجية. ومثلما لاحظنا في الفصل الأول، فإن المزيد من الموارد قد تمّ استهلاكها في مرحلة الصيانة، بدلاً من أي مرحلة أُخرى من دورة حياة البرنامج. فالنظم البرمجية الضخمة لا تخمد؛ ولكن تتغير ببساطة.

ويمثل التغير عاملاً ثابتاً في تطوير البرنامج، حيث يجب أن نملك مجموعةً من الأهداف، التي تتجاوز تأثير التغيرات. فقد وصف كل من C.A.Irvine و J.B.Goodenough و O.T.Ross، في أبحاثهم الشهيرة، هذه الأهداف قائلين: " يوجد أربع صفات كافية، لتكون مقبولةً كأهدافٍ من أجل كل علم هندسة البرمجيات. هذه الصفات، هي قابلية التعديل، والفعالية، والوثوقية، وقابلية الفهم".

قابلية التعديل (Modifiability):

تمثل قابلية التعديل هدفاً صعب المنال والقياس. وبشكلٍ أساسي، لأنها "تؤدي لتغيرات متحكم بها. فبعض الأجزاء أو الهيئات فيها، تبقى نفسها، بينما تتغير البقية. وكل ذلك بطريقةٍ تسمح بالحصول على نتيجة جديدة نرغبها". ويمكننا تغير نظامٍ برمجي، لأحد السببين التاليين: الأول، الإجابة على تغيرٍ في متطلبات النظام، والثاني، تصليح خطأٍ أدخلناه مسبقاً في معالجة التطوير.

وعندما نصمم برنامجاً، بطريقة أو بأخرى، يجب علينا ضبط بنية التصميم، بطريقةٍ واضحةٍ ومتماسكة. وبشكلٍ عام، فبما أنّ معظم لغات البرمجة ليست مقروءة بشكلٍ جيد، فهذا يجبرنا على استخدام وثائق خارجية لعكس هذه البنية، لكننا نرغب في الواقع، المحافظة على تصميمنا في البرنامج نفسه.

ومن أجل تغيير النظم بشكلٍ فعلي، يجب أن نأخذ بالحسبان، جميع قرارات التصاميم الضمنية والصريحة، والتي تشكل جزءاً من الحل. وبشكلٍ آخر، نحن مجبرون على ترميم برنامجاً بشكلٍ مستقلٍ عن التصميم الأصلي. وهكذا، نستخلص البنية

المنطقية لبرنامجنا. بعد عدة تكررات، تُصبح البنية الأصلية غامضة، مما يجعل البرنامج أكثر صعوبةً. فإذا كانت النظم البرمجية قابلةً للتغيير، يجب أن يكون إدخال تغيرات دون زيادة تعقيد النظام الأصلي، ممكناً.

الفعالية (Efficiency):

إن الفعالية تعني، أنّه يجب استثمار نظامٍ برمجيٍ، باستخدام مجموعةٍ جاهزةٍ من الموارد، بطريقةٍ أمثلية. ويمكننا تصنيف هذه الموارد، في مجموعتين: الموارد الزمنية، والموارد الحجمية. ويمكن أن نملك موارد زمنية محدودة، إذا وجب تنفيذ إجراء مهمةٍ، ضمن مدة محددة. كمثل أخذ عيناتٍ من حساسات، أو الاستجابة لمقاطعةٍ خارجية. وبشكل واضح، ترتبط الموارد الزمنية وبقوة، بالبنية الصلبة الأساسية، بالرغم من أن اختيارنا للخوارزميات البرمجية، سيؤثر بالتأكيد على زمن التنفيذ الكلي. ومن جهةٍ أخرى، ترجع الموارد الحجمية للجوانب الفيزيائية للحل، مثل عناوين الذاكرة، أو عدد أجهزة الطرفيات الجاهزة.

وغالباً، يجب أن تأخذ تطبيقات النظم المحمولة، صفي الموارد بالحسبان. فإذا وجب على النظام الإستجابة لحدثٍ حقيقي، عندها يصبح استخدام الموارد الزمنية حرجاً. ومن جهة أخرى، إذا قُيدت البنية الصلبة الأساسية بحجم فيزيائي، أو بحدود قدرة، مثلاً في قمرٍ صناعي، أو في سيارة، عندها تكون الموارد الحجمية ضرورية للحل. وفي كثيرٍ من الحالات، من غير الممكن استخدام كلا الموردين، في نفس الوقت، وبشكل فعال. ولذلك، يجب إيجاد حلٍ وسط لهما. وهذا هو جوهر الهندسة، لكل نظامٍ من العالم الحقيقي.

ومن الواضح، أنها ليست فقط النظم المحمولة هي النظم الوحيدة، التي من أجلها يجب أن يقلق المطورون على الفعالية. فكلما أصبح الحاسوب ذا مقدرةٍ أكبر، كلما تزايدت الحاجة للإستراتيجيات الفعالة، في جميع مجالات المسائل. فكل من التعرف على الأشكال، وتخطيط الشبكات، وخوارزميات البحث، تتطلب الفعالية أيضاً.

وغالباً ما نهتم باكراً بموضوع الفعالية، أثناء عملية التطوير. ونتيجة لذلك، غالباً ما نركّز على الفعاليات الصغيرة، بدلاً من الفعاليات الكبيرة. وبالتالي، يجب أن نعرف أنّ "بصيرةً ممتازةً، تعكس فهماً كبيراً للمسألة، لها أثر على الفعالية، أكبر بكثيرٍ من أية تسليةٍ بدون فائدة في البتات (bits)، ضمن بنيةٍ ضعيفة ".

الوثوقية (Reliability) :

تمثل الوثوقية هدفاً حرجاً لأي نظام معلوماتي، يجب أن يُستثمر لفترة زمنية طويلة دون أي تدخل إنساني. وأكثر من ذلك، إذا تحكم ذلك النظام بموردٍ حرج، مثل مصنع للطاقة الذرية، أو نظام ملاحةٍ لسفينةٍ فضائية، فإنّ تكاليف الإخفاق تكون عالية جداً، لأننا سمحنا لأنفسنا بإهمال الوثوقية. حيث "يجب أن تمنع الوثوقية كلاً من عيوب التصميم والدراسات، وتسمح باستعادة أعطال وعيوب الأداء".

ومثلما عرّفنا الوثوقية، يبدو أنّ هذا الهدف يجب أن يكون حاضراً طيلة دراسة وتصميم البرنامج. حيث "لا يمكننا صياغة الوثوقية إلا من البداية؛ ولا يمكن إضافتها في النهاية". ولا يمكننا، على أي حال، أن نتوقع وثوقيةً كاملة، لأنّه سيكون هنالك حتماً ظروف غائبة عن مراقبتنا، مثل أعطال مأساويةٍ في البنية الصلبة، التي تؤثر حتى في النظم الإحتياطية. ومهما كان نوع الخطأ، متوقعاً أم لا، فإننا نأمل أن يتراجع نظام موثوق باعتدال، دون أن يسبب أي تأثيراتٍ جانبيةٍ خطيرة، مثل ذوبان قلب مفاعل، أو فساد نظام ملاحة قمرٍ اصطناعي.

قابلية الفهم (Understandability) :

يمكن أن يكون موضوع قابلية الفهم، الأكثر حرجاً لمساعدتنا على إدارة تعقيد نظمنا البرمجية. وتمثل قابلية الفهم جسراً بين فضاء مسألتنا الخاصة، والحل الموافق. بمعنى آخر، لكي يكون النظام مفهوماً، يجب أن يكون نموذجاً دقيقاً لرؤيتنا للعالم الحقيقي. فإذا توجب علينا تطبيق معرفتنا على مسألةٍ صعبة، يجب أن نبني حلاً ذا بنيةٍ فعليةٍ ومميزة. ومثلما ذكرنا سابقاً، إن إيجاد بنية كهذه في البرنامج نفسه، هو أمر جوهري، لتكون نظمنا قابلةً للتغير، وفعالةً، وموثوقة.

وسيكون نظام ما قابلاً للفهم، بسبب عدة عوامل من عدة مستويات. ففي أخفض مستوى، يجب أن يكون الحل البرمجي قابلاً للقراءة، كنتيجةٍ لأسلوبٍ ترميزٍ مناسب. وفي أعلى مستوى، يجب أن نكون قادرين على أن نعزل بسهولة، بُنى المعطيات (الأغراض)، والخوارزميات (العمليات) في الحل، والتي توافق معطيات وخوارزميات العالم الحقيقي. ومثلما سنرى، ترتبط وبقوة قابلية الفهم، بلغة البرمجة التي نستخدمها كوسيلةٍ للتعبير عن الحل.

٢ ـ ٢ ـ مبادئ هندسة البرمجيات

(Principles of Software Engineering):

إن الأهداف التي ناقشناها في المقطع الأخير، تنطبق على أي نظامٍ برمجي، ضخماً كان أم صغيراً. وعلى أية حال، لا يمكننا الإكتفاء بمعرفة هذه الأهداف، وبشكلٍ سلبي، واستخدام طريقة تطويرٍ غير أكاديمية وصعبة فيما بعد، ونأمل أن نصل لهذه الأهداف. ولكن على العكس، عندما نصمم برنامجاً، يجب أن نطبق مبادئ أسسٍ هندسيةٍ جيدة للوصول لهذه الأهداف، بما في ذلك:

- التجريد (Abstraction).
- إخفاء المعلومات (Information hiding).
- الوحدوية (Modularity).
- المحلية (Localization).
- التجانس (Uniformity).
- التكاملية (Completeness).
- الصلاحية (Confirmability).

وفي المقاطع التالية، سنرى كيف يمكن أن يقود تطبيق هذه المبادئ لحلولٍ قابلةٍ للتغير، فعالة، موثوقة، وقابلة للفهم.

التجريد وإخفاء المعلومات (Abstraction & Information Hiding) :

لقد اعترفنا منذ قليل، بأنّ تعقيد البرنامج يشكل السبب الرئيسي لمسائلنا. ويمثل *التجريد* أحد المبادئ الأساسية لإدارة هذا التعقيد. ولا يمثل التجريد مفهوماً جديداً؛ إذ أننا نستخدمه في أي شيء نعمله. فعلى سبيل المثال، يمكن أن يتعامل برنامجنا مع سواقة أقراص، وبالتالي، يمكننا رؤية هذه الأداة الفيزيائية، من عدة وجهات نظر:

- كمجموعةٍ من الملفات المنطقية.
- كوسيلة تخزين ذات سعةٍ كبيرة، منظمةٍ في مسارات وقطاعات.
- كمجموعةٍ من وسائل التخزين الثنائية المعنونة.
- كوسيلةٍ فيزيائية، بحاجة لتحكم وإشارات معطيات.

ولقد شكلنا هنا، سلماً للتجريد، يكون فيه كل مستوى تجريد، مبنياً على المستويات الأدنى. ففي برنامجنا، يمكن أن نختار مستوى التجريد المناسب لاحتياجاتنا. وعلى سبيل المثال، في حال نظام إدارة قاعدة معطيات، نحتاج أن نرى سواقة الأقراص فقط، كأنها مجموعة من الملفات المنطقية. فإذا كتبنا محرك سواقةٍ لنظام استثمارنا، يجب علينا أخذ رؤيةٍ مختلفة من نفس الأداة الفيزيائية، وقد يكون في مستوى إشارات التحكم والمعطيات."ويتمثل جوهر التجريد، باستخراج الخواص الأساسية، بإهمال التفصيلات غير الأساسية"، مثلما فككنا حلنا إلى أجزاء مختلفة، فكل وحدة بالتفكيك، تصبح جزءاً من التجريد في مستوى محدد. وأكثر من ذلك، يمكننا تطبيق التجريد في حلولنا على كلٍ من المعطيات، والخوارزميات.

ويمكن أن يحدد كل مستوى تجريد، أنواع معطيات مجردة (مثل الملف، التسجيلة، أو قطاع من القرص)، وكل واحدةٍ فيها، موصفة بمجموعة من القيم، ومجموعة من العمليات، التي يمكن تطبيقها على كل غرضٍ من النوع. فعلى سبيل المثال، يمكن أن يُحتاج إلى نظام قواعد المعطيات، ليعالج مجموعة ملفاتٍ منطقية، بينما محرك نظام الإستثمار، سيرى سواقة القرص، وكأنّها كتلة معنونة من الكلمات. ومن الواضح أنه، إذا كتبنا نظام قواعد المعطيات، فيجب ألا نشغل أنفسنا بالتنظيم

الفيزيائي للكلمات المخزنة على القرص، ولا نقلق حول الملفات المنطقية، إذا كتبنا محرك السواقة.

يمكن أن نقوم بعملٍ مشابهٍ في التجريد الخوارزمي. فنظام قاعدة معطياتنا سيحتاج *لفتح*، *وإغلاق*، *وقراءة*، *وكتابة* هذه الملفات المنطقية، وبالتالي، لا نريد أن نشغل أنفسنا بالتفاصيل الفيزيائية، من اختيار مسار، وإيجاد المقطع الملائم، وتحقيق ترميز الحشو، ومن ثمّ، إيجاد قطعةٍ من ملفٍ مرغوبٍ من جديد. وبدلاً من ذلك، نرى كلاً من *الفتح*، *والإغلاق*، *والقراءة*، *والكتابة* كعملياتٍ مجردة، دون القلق حول تفاصيل تنفيذها. وبالطبع، ففي آخر الأمر، يجب أن نشغل أنفسنا بهذه التفاصيل، ولكن يمكننا تأجيل التنفيذ للمستويات الأكثر انخفاضاً من حلولنا. وبهذه الطريقة، نقلّص عدد الكيانات التي نريد أن نحجزها بالمستوى الحالي من التفكيك.

وفي هذا المثال الأخير، لقد طبقنا فعلياً مبدأً ثانياً من هندسة البرمجيات، يُدعى *إخفاء المعلومات*. ولما كانت التجريدات تستخرِجُ التفاصيل الأساسية لمستوى محدد، "... فإن الغاية من الإخفاء، تتمثل بعدم إمكانية الوصول لبعض التفاصيل، التي يجب ألا تؤثر على بقية أجزاء النظام". وفي مثالنا، نظام قاعدة المعطيات، سيُخبرُنا مبدأ إخفاء المعلومات، بأنّه يجب ألا نسمح لمبرمجٍ بالوصول إلى التفاصيل الفيزيائية لسواقة القرص، بالكتابة المباشرة مثلاً" على مقطعٍ محدد. ولإجازة هكذا أفعال، سنسمح للمبرمج بالتعدي على تجريدنا المنطقي لملف القرص.

وبالنتيجة، فإن إخفاء المعلومات يحذف طريقة تنفيذ الغرض أو العملية، وهكذا، يركز انتباهنا على التجريد من المستوى الأعلى. وأكثر من ذلك، عندما نُخفي قرارات التصميم ذات المستوى المنخفض، مثل التنظيم الفيزيائي للملفات على قرص، فإننا نمنع بذلك إستراتيجيات المستوى الأعلى، من تأسيسها على التفاصيل ذات المستوى المنخفض. وبالتالي، في مثالنا، إذا غيرنا سواقات الأقراص لتقديم سعة تخزين أكثر، فلن تؤثر إعادة التنظيم الفيزيائي للقرص، على تجريدنا عالي المستوى لنظام الملف المنطقي.

وتنتج الفوائد من تطبيق التجريد وإخفاء المعلومات، بتطبيقها تقريباً على كل أهداف هندسة البرمجيات. وإذ يساعد التجريد على الصيانة، وقابلية الفهم للنظم، وذلك بتقليص التفاصيل التي يحتاج المطور إلى معرفتها في كل مستوى. وأكثر من ذلك، إننا نحسن وثوقية النظم عندما لا نسمح، في كل مستوى تجريد، إلاّ لبعض العمليات، ونمنع العمليات التي تتعدى على رؤيتنا المنطقية لذلك المستوى.

الوحدوية والمحلية (Modularity & Localization) :

وهناك أداة أساسية أخرى، تساعدنا على إدارة تعقيد النظم البرمجية، تتمثل *بالوحدوية*. " وتبحث الوحدوية بالطريقة، التي من أجلها تُسّهل بنية الغرض، الحصول على بعض الأهداف. وتتمثل الوحدوية بأنها "بنية غرضية". وبالتال،ي تطبق على البنية الفيزيائية لنظمنا.

فإذا طبقنا طريقة التصميم البرمجي التنازلية، من الأعلى للأدنى، سنحلل بشكل نموذجي، كلاً من المستويات المتتالية، إلى وحداتٍ وظيفيةٍ مختلفة. وبشكل عام، فإن الوحدات البرمجية عالية المستوى، توافق تجريداتنا عالية المستوى، وهي إذاً، غير مرتبطة نسبياً بالآلة. وأكثر من ذلك، ستحدد الوحدة العالية أي فعلٍ سيؤخذ، بينما تعِّرف الوحدات منخفضة المستوى، كيف سينفذ كل فعل. فإذا استخدمنا طريقة التصميم التصاعدية (من الأدنى للأعلى)، فإن الميزات ذاتها تطبق في كل مستوى، ولكن بدلاً من تحليل نظامنا (تنازلياً من الأعلى للأدنى)، فإننا نبني النظام صعوداً، اعتباراً من وحداتٍ منخفضة المستوى، باتجاه وحداتٍ أكثر تعقيداً.

ومن المهم أن نعرف، أنّ نظم الإنتاج نادراً ما تُبنى كاملاً بطريقة من الأعلى للأدنى، أو بطريقة من الأدنى للأعلى. بينما، أي نظامٍ معقدٍ فعلياً، يستخدم على الأرجح كلا الطريقتين. إن نظاماً ضخماً، يكون عادة، محللاً من الأعلى للأدنى. ويكون غالباً، مركباً من مركبات برمجية، يمكن إعادة استخدامها - بمعنى آخر، إن النظام الضخم، يكون مركباً عادة، من الأدنى للأعلى.

وبطريقةٍ أكثر شمولية، يمكن أن تكون الوحدات وظيفية (إجرائية موجهة)، أو تصريحية (غرض موجه). فإذا أردنا إدخال الوثوقية في بناء نظامنا، يجب أن نقدم واجهة تخاطبٍ معرَّفةٍ بشكلٍ جيد، لكل وحدة. وعلى أية حال، ومهما تكن الحالة التي يتم بها تعريف وحدة برمجية، فإنها ستتفاعل بالتأكيد، مع وحدات أُخرى؛ وهكذا، نعرّف الإقتران، بأنه "قياس شدة الإرتباط بين الوحدات". وبشكلٍ مثالي، نريد أن تكون وحداتنا البرمجية ضعيفة الإقتران، لتسمح لنا بمعالجة كل وحدةٍ بطريقةٍ مستقلةٍ نسبياً عن بقية الوحدات. ويمكننا تطبيق قياسٍ آخرٍ على الوحدات، يُدعى *التماسك*، الذي يعرف " إلى أي حدٍ تكون العناصر الداخلية مرتبطة، أو تمّ إعادة ربطها مع بعضها البعض (داخل الوحدة)". وفي هذه الحالة، نريد وحداتٍ ذات تماسكٍ قوي، بحيث تكون مكونات وحدة محددة، مرتبطة وظيفياً ومنطقياً.

ويساعدنا أيضاً تطبيق مبدأ *المحلية*، بخلق وحداتٍ ضعيفة الإقتران، وقوية التماسك. ويخص التماسك بشكل أساسي، القرب الفيزيائي. وبشكلٍ مثالي، نريد تجميع موارد حسابية، مرتبطة منطقياً في وحدة فيزيائية واحدة، حتى يقودنا هذا إلى وحدةٍ متماسكةٍ بقوة. وأكثر من ذلك، تؤدي المحلية على أنه إذا كان هناك وحدة محددة مستقلة بشكل كافٍ، فإن هذا يقودنا إلى تنظيمٍ ضعيف الإقتران.

ومن وجهة نظر أهداف هندسة البرمجيات، فإن مبدأ الوحدوية والمحلية، يدعمان مباشرة قابلية التعديل، والوثوقية، وقابلية الفهم. فإذا كان لدينا نظام تمّ بناؤه بشكلٍ جيد، يجب أن نكون قادرين على فهم أي وحدة، بطريقةٍ مستقلةٍ نسبياً عن الوحدات الأُخرى. وأكثر من ذلك، عندما نمركز قرارات تصاميمنا في وحدات محددة، فإنه يمكننا تقيد تأثيرات التعديل، على مجموعةٍ صغيرةٍ من الوحدات. وأخيراً، إذا استخدمنا وحدوية غرضية، فإننا سنحد من علاقات الإرتباط بين وحدات البرنامج، محسّنِين بذلك وثوقية برنامجنا.

التجانس، والتكاملية، والصلاحية

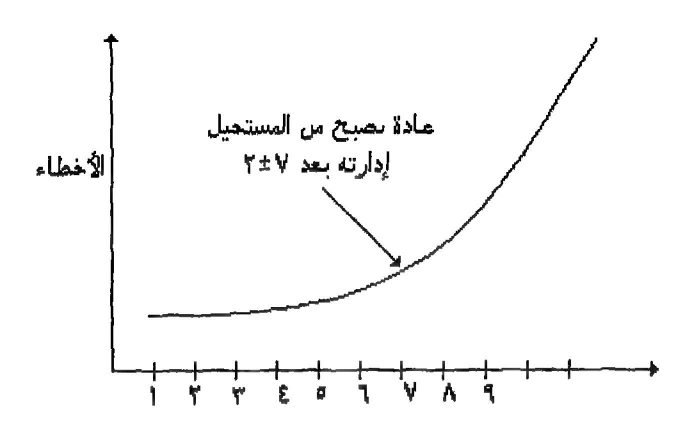
(Uniformity, Completeness, & Confirmability):

من المحتمل أن يكون التجريد والوحدوية، هي المبادئ الأكثر أهميةً، والتي يمكن أن نستخدمها للتحكم بتعقيد البرنامج. ومع ذلك، فإن هذه المبادئ ليست كافية، لأنها لا تكفل الوصول إلى نظمٍ صحيحةٍ أو متماسكة. ولذلك، يجب أن نطبّق مبادئ التجانس، والتكاملية، والصلاحية، للحصول على هذه الميزات.

ويدعم مبدأ التجانس، مباشرةً، هدف قابلية الفهم. ويعني *التجانس* ببساطة أن الوحدات تستخدم رموزاً متماسكة، وهي حرة من أي اختلافات غير ضرورية. وينتج التجانس عادةً، من أسلوب ترميزٍ جيد، نطبق ضمنه بنية تحكم متماسكة، وسلسلة استدعاءات لعملياتنا، حيث تكون الأغراض المرتبطة منطقياً، هي ذاتها في جميع المستويات. وفي فصولٍ لاحقة، وفي الملحق D، سنقترح أسلوب ترميزٍ كهذا، عن طريق أمثلة.

إن مبدأ التكاملية والصلاحية، يدعمان أهداف الوثوقية والفعالية، وقابلية التعديل، ويقوم بمساعدتنا على تطوير حلولٍ صحيحة. بينما يستخرج التجريد التفاصيل الأساسية لكيانٍ محدد، وتضمن *التكاملية* ظهور جميع العناصر الهامة. وبمعنى آخر، فإن التجريد والتكاملية، يساعدانا على تطوير وحداتٍ ضرورية وكافية. هذا، وقد تحسنت الفعالية ما دمنا، نستطيع الآن ضبط تنفيذٍ منخفض المستوى، دون التأثير على أي وحدةٍ عالية المستوى.

وبالإقتران مع التكاملية، فإن مبدأ الصلاحية، *ينضوي على* أنّه يجب علينا تحليل نظامنا بطريقة يسهل فحصها، وهذا ما يساعدنا على جعل نظامنا قابلاً للتغيير. وليس من السهل تطبيق مبدئي الصلاحية والتكاملية. فبعض الأدوات، مثل لغات البرمجة ذات النوعية القوية، تسمح لنا بالحصول على نظمٍ صالحة، لكن، مثلما سنرى، يجب تطبيق مبادئ هندسة البرمجيات، لنتأكد من أنّ نظامناً كامل وصالح.



الشكل ٢ ـ ١. عدد عناصر المسألة، التي يجب أن تُعامل معاً.

٢ ـ ٣ ـ طرق تطوير البرمجيات (Approaches to Software Development):

قلنا في الفصل الأول، أنّه توجد قيود إنسانية أساسية، لقدرتنا على إدارة عددٍ من الأغراض أو المفاهيم المختلفة، في وقتٍ واحد. ففي عام ١٩٥٤، استنتج عالم النفس George Miller، بأنّ القيود لعددٍ من كيانات إنسانية، التي يمكن أن تعالج بنفس الوقت، تمثل تقريباً سبعة، بإضافة أو نقصان اثنين. ومثلما نرى في الشكل (٢ ـ ١)، الذي اشتق من عمل Miller، فإنّ مقدرتنا لإدارة هذا التعقيد، تنهار بعد هذا العدد.

وإن تطوير النظم البرمجية، يمثل نشاطاً لحل المسائل، وبالتالي، فإن قيد السبعة، بإضافة أو نقصان اثنين، يبدو مطبقاً. ونقترح بأنّه يمكن لمبادئ هندسة البرمجيات، أن تساعدنا على تحليل النظم بطريقة ما، في أي مستوى من حلولنا، وعدد الكيانات التي يجب أن نهتم بها، ستبقى ضمن هذه الحدود. وفي هذا المقطع، سنفحص طرق تصميم برمجياتٍ مختلفة، وتقنيات إدارة تنفيذ هذه المبادئ، وبهذا، تساعدنا على إدارة تعقيد برمجتنا.

طرق التصميم (Design Methods):

لا يمكن تطبيق مبادئ هندسة البرمجيات بنجاحٍ ضئيل ، بل يجب أن نبني نظمنا بطريقةٍ منظّمة. والأكثر أهميةً، هو أنه عندما نقسم مسألتنا لوحدات، يجب أن نستخدم معايير متماسكة لتحليلها. وعادةً تجسّد هكذا معايير، بما ندعوه *طرق* تطوير البرمجيات. وعلى وجه التخصيص، فإن الطريقة، هي معالجة منظمة لإنتاج منتجاتٍ برمجية. وتختلف الطرق عن *المنهجيات*، والتي هي مجموعة من الطرق المتعاونة. وبمعنى أكثر شمولية، هي طريقة فلسفية للتطوير البرمجي. فهي تقدم هيكلاً لتطبيق الطرق – وإستراتيجيةً لقيادة جميع مراحل دورة حياة التطوير البرمجي.

والطرق ضرورية لعدة أسباب، وعلى رأسها إدارة التعقيد. وبالإضافة لذلك، فإن الطرق تُحسن الإتصال بين المطورين، وتجمع مراحل تطوير دورة الحياة. والإتصال والإنتقال بهدوء بين المراحل، تعتبر هامةً بشكلٍ خاص، للنظم البرمجية الضخمة، والتي تشرك عادةً عشرات المطورين المنفصلين بالزمن، عندما تكون دورات الحياة طويلةً، وبالحجم، عندما تكون المواقع متباعدةً.

وفي الفصل ١٩ ، سنفحص المراحل التقليدية لدورة حياة تطوير برنامجٍ بالتفصيل. حالياً، فدعنا نركز على فئةٍ هامةٍ من الطرق، التي تبحث في مراحل البنية والتصميم المفصل، للتطوير البرمجي.

ويمكننا تقريباً، تقسيم كل طريقة تصميمٍ برمجي موجودة، إلى واحد من الصفوف الثلاثة التالية:

- تصميم بنيوي، من الأعلى للأدنى (تنازلي).
- تصميم بنى معطيات.
- تصميم غرضي التوجه.

وإن المناقشة الكاملة لكل صف، تخرج عن إطار هذا الكتاب، ولكن في الفقرات القليلة التالية، سنقدم التفاصيل الأساسية لكل واحد منها.

تركز طرق التصميم البنيوية من الأعلى للأدنى، على التجريد الخوارزمي للمسألة. ولهذا السبب، غالباً ما ندعو هكذا طرقاً بأنها طرق *مقادة بالمعالجة* (Process driven). وقد تجسدت الخصائص الأساسية للطرق المقادة بالمعالجة، بواسطة أعمال Constantine، والذي إقترح بأن نعمل وحدة، لكل خطوة في المعالجة الإجمالية. وقد شارك أيضاً Yourdon بشكل أساسي، في تبسيط التصميم البنيوي من الأعلى للأدنى. وبإتباع هذه الطريقة، نحصل على وحداتٍ وظيفيةٍ عاليةٍ للبرامج. وإن طريقة التصميم من الأعلى للأدنى، متأثرة بشدة بتوبولوجيا لغة FORTRAN (كما سنرى في المقطع التالي)؛ وللأسف؛ فإن طريقة التصميم من الأعلى للأدنى، لا تنطبق مباشرةً على المسائل التي تستخدم التوازي. في المستوى الأعلى لحلنا، نعين أعلى مستوى من التجريد الخوارزمي (الـ "ماذا" من المعالجة)؛ وتقدم المستويات المنخفضة عملياتٍ بدائية، تنفّذ هذه الأفعال عالية المستوى. وغالباً، ما تتصف بنية النظام *بمخطط بنية*، يوضح الإرتباطات الهرمية بين الوحدات الوظيفية.

وإن تصميم *بنى المعطيات*، مثلما توحي التسمية، يركّز على معطيات المسألة. ولقد كان كل من D.Jackson و P.Warnier الرواد الأوائل لطريقة بنى المعطيات، التي تمّ برهان فعاليتها في تطبيقاتٍ نموذجيةٍ لـ COBOL . وباستخدام هذه التقنية، نعين في البدء، بُنى معطياتنا، بعد ذلك نبني بنية النظام، المؤسسة على بُنى المعطيات. ووفق هذه الطريقة، نحاول تعريف تنفيذ الأغراض في فضاء حلنا بوضوح، ومن ثمّ نجعل بنية هذه الأغراض مرئية للوحدات الوظيفية، الضرورية لتقديم العمليات على هذه الأغراض.

وهنا ، يجب أن نذكر تأثير Parnas، الذي عمل في جامعة North Carolina، وكان لأعماله تأثير كبير على طرق التصميم. فقد أقترح Parnas بأنّ نحلل نظمنا، بطريقةٍ تخفي فيها كل وحدة التصميم. وطريقته ليست مقادة بالمعالجة، ولا بالمعطيات، ولكن، تصلح لالتقاط خيارات تصاميمنا في أخفض مستوى ممكن. وهذا هو جوهر إخفاء المعلومات.

وكان لعمل Parnas، تأثير كبير على الصف الثالث من طرق التصميم، *التصميم غرضي التوجه*. وسنفحص هذه الطريقة بالتفصيل، في الفصل القادم. وباختصار، تركز الطريقة على الأغراض، كوسائل أساسية مستخدمة في الحسابات؛ ولهذا السبب، يتم تنظيم بنية نظام حول مجموعة أغراض - وليس تجريدٍ خوارزمي. وبهذه الطريقة، نجمع كل صفٍ من المعطيات، والعمليات المرتبطة، في وحدةٍ وحيدة. ولم تولد هذه الطريقة، فقط على يدي Parnas، ولكن أيضاً على أيدي B.Liskov و J.Guttag، من المعهد التكنولوجي في Massachusetts و Robinson و K.Leavitt من معهد البحوث في Stanford. وبالإضافة لذلك، تتبع هذه الطريقة، طريقة التوجه الغرضي، المفضلة من قبل اللغات Smalltalk, SIMULA67, C++, Object Pascal, CLOS (Common Lisp Object System), And ADA 9X. وإن التقنية الخاصة التي قدمناها للتصميم غرضي التوجه، لبرنامج ADA، تمّ إبداعه من قبل R.Abbott من جامعة ولاية كاليفورنيا في Northridge.

مشاكل الإدارة (Management Issues):

مهما كانت طرق التصميم التي نستخدمها، يجب أن نتذكر دائماً، أنّ القدرات الإنسانية التي تدير التعقيد، محدودة. وبما أنّ علم الحاسوب لم ينتشر للحد الذي يكون فيه التوليد الآلي للبرامج، يمثل حقيقة عملية، فإننا يجب أن نطبق تقنيات الإدارة، لدعم طرق تصميمنا. والمناقشة الكاملة لهذه التقنيات، تخرج عن نطاق هذا الكتاب، ولم يقدم أيضاً هذا النص، إلا مراجع، من أجل معلوماتٍ إضافية.

ولا تختلف إدارة تطوير برنامجٍ، عن أي جهدٍ هندسيٍ معقد، ماعدا، أن المشاريع (البرمجية) تكون أقل مادية من، لنقل، جسرٍ أو سفينة. وفي معظم الحالات، يكون المشروع البرمجي معقداً كثيراً، ووحيداً. وفي الفصل ١٩، سنناقش طريقة دورة حياة تطويرٍ برمجيٍ في ADA؛ وهنا سنعرف ببساطة، بأنّ طرق التصميم لا تُطبّق، إلا على جزءٍ واحدٍ من دورة حياةٍ كاملة، بينما يجب أن نطبق مبادئ الإدارة العملية، خلال جميع مراحل التطوير البرمجي.

وكخطوةٍ أولى، من الضروري أن نشكل نموذجاً كاملاً وصحيحاً لفضاء المسألة. ومن أجل هذا، يمكننا تطبيق *التحليل البنيوي، وتقنية التصميم* (*Structured Analysis & Design Technique*) (SADT)، لخلق هذا النموذج. وعند تحليل نظامنا، يمكننا اختيار توثيق تصميمنا، باستخدام ***مخططات تدفق المعطيات***. وللوصول إلى العناصر الوظيفية من تصميمنا، يمكننا استخدام ***المخططات البنيوية***. ويمكننا كذلك، إدخال تصميمٍ مفصل على شكلٍ من PDL، الذي نستخدم فيه لغة برمجةٍ (مثل ADA)، لتوثيق القرارات المفصلة لتصميمنا. وتمثل ***المراجعة البنيوية***، أداةً طرفيةً قويةً للإدارة، التي تستخدم تفتيشاً منمقاً لمركبات البرنامج. وتقدم المراجعة البنيوية، خلاصة التطورات التي طرأت على تطور البرنامج، وآلية لتحقيق النوعية والتكاملية للمنتج.

٢ ـ ٤ ـ اللغات وتطوير البرمجيات (Languages&SoftwareDevelopment):

إن الطرق لوحدها، ليست كافيةً لخلق حلولٍ حاسوبية. ويجب أن نملك الوسيلة المناسبة، بشكلٍ عام، على شكل لغات برمجة، للتعبير عن تصاميمنا، وتنفيذها. وفي اللائحة التالية، صنف P.Wegner، بعضاً من معظم اللغات الشائعة، مقسمةً على أجيال، ومرتبةً وفقاً لتاريخ تقديمها:

- **لغات الجيل الأول (١٩٥٤ ـ ١٩٥٨):**

FORTRAN I
ALGOL 58
Flowmatic
IPL V

وتعتبر هذه اللغات، من أجل التعابير الرياضية.

- **لغات الجيل الثاني(١٩٥٩ ـ ١٩٦١):**

FORTRAN II	برامج جزئية، ترجمة منفصلة
ALGOL 60	بنية كتلة، أنواع معطيات
COBOL	توصيف معطيات، معالجة ملفات
LISP	معالجة لوائح، المؤشرات

- **لغات الجيل الثالث (١٩٦٢ ـ ١٩٧٠):**

FORTRAN+ALGOL+COBOL	PL/1
خَلَف صارم لـ ALGOL 60	ALGOL 68
خَلَف بسيط لـ ALGOL 60	Pascal
الصفوف، تجريد المعطيات	SIMULA

- **فجوة الأجيال (١٩٧٠ ـ ١٩٨٠):**

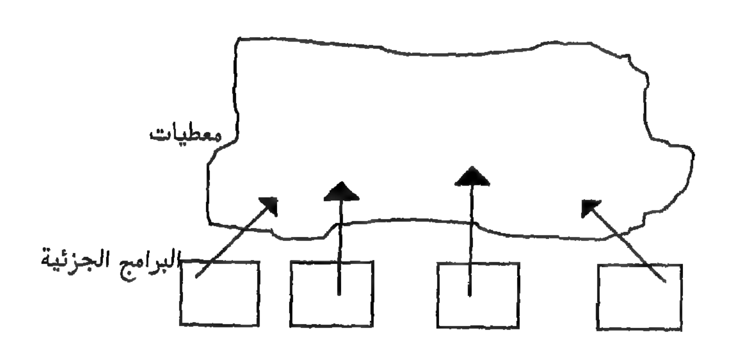
هناك الكثير من اللغات المختلفة، ولكن لم تثبت أية واحدة.

ومثلما نرى في هذه اللائحة، فإن اللغات الأكثر استخداماً، تتمثل باللغات عالية المستوى (المعروفة بتنويعات من FORTRAN و COBOL)، والتي تمّ خلقها في بداية تاريخ علم الحاسوب، والتي سبقت كثيراً فهمنا لمشاكل تطوير البرمجيات الهامة. وكنتيجةٍ لذلك، لا تعكس هذه اللغات طرق التصميم المعاصرة. وهكذا، يجب أن ننمق كل لغة بمعالجٍ أولي مثل S-FORTRAN ولاحقاته (مثل FORTARN 77)، لإجبارها على ملاءمة الطرق العصرية. في أية حالة، فقد تمّ تطوير هذه اللغات عالية المستوى، في زمنٍ كانت فيه المسائل بسيطة نسبياً، مقارنةً بمجالات التطبيقات الحالية.

ومازالت كلٌّ من FORTRAN و COBOL، ملائمتين لمجالات تطبيقاتهن الخاصة. وعلى أية حال، فمنذ تطويرهن، ظهر مجال النظم البرمجية الضخمة جداً. ولم تصمم FORTRAN، ولا COBOL ولا معظم لغات البرمجة عالية المستوى الأخرى، لتعالج تعقيد هذا المجال، وهي أقل بكثير من أن تواجه الصعوبات المرتبطة بتطوير برمجياتٍ مؤلفةٍ من آلاف إلى ملايين أسطر الترميز.

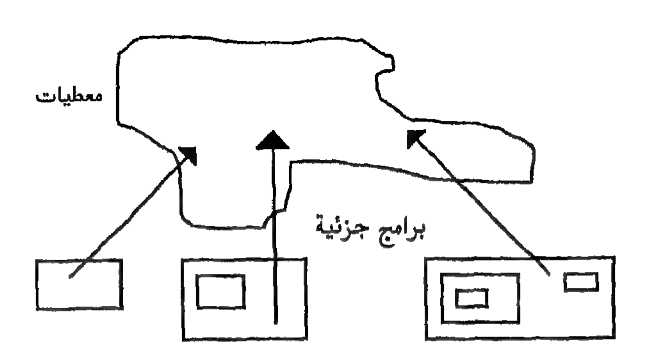
فإذا فحصنا تبولوجيا هذه اللغات المبكرة، يمكننا أن نبدأ بفهم بعض مسائلها المتأصلة. وكما يشير الشكل (٢ ـ ٢)، فإن لغات الجيل الأول والثاني، مثل COBOL وشكلي FORTRAN، تبدي بنيةً مسطحةً نسبياً، مؤلفةً من معطياتٍ

عامة، ومستوى واحدٍ من البرامج الجزئية. وتشير الأسهم في الشكل، إلى ارتباطات البرامج الجزئية نحو المعطيات. خطأٍ في قطاعٍ واحدٍ من برنامج، يمكن أن يسبب خللاً هائلاً في بقية النظام، بسبب بنى المعطيات العامة. وأكثر من ذلك، عند إجراء تعديلات على نظامٍ ضخم، فمن الصعب جداً إن لم يكن مستحيلاً، صيانة طراز بنية التصميم الأولي. وحتى بعد فترةٍ قصيرةٍ من الصيانة، يحتوي البرنامج المكتوب بإحدى هذه اللغات، على قسمٍ ضخمٍ من الإقتران المتصالب بين وحدات البرنامج، مُعرِّضاً بذلك وثوقية النظام الكلي للخطر، ومعيقاً إعادة استخدام البرنامج، ومقلصاً وضوحية الحل.



الشكل ٢ ـ ٢. طبولوجيا لغات الجيل الأول والثاني.

ومع تطوير اللغات مثل ALGOL 60 في الجيل الثاني، وبعض لغات الجيل الثالث. كنا قادرين على تقديم بنيةٍ كثيرة التداخل لخوارزمياتنا، بالرغم من وجود قليل من التحسين في الوظائف، لوصف بُنى المعطيات. وكما هو مبين في الشكل ٢ ـ ٣. فقد كانت توبولوجيا هذه اللغات تختلف، بشكلٍ طفيفٍ عن تبولوجيا الجيل السابق. وهكذا، فقد عانت من نفس المشاكل. والقليل من هذه اللغات قد تمّ تطويرها خلال هذه الفترة الزمنية، والتي قدمت بالفعل بنية معطيات أفضل. هذه اللغات هي: LIS &. و CLU. و Alphard، و SIMULA. لكن، و لا لغة من هذه اللغات. لاقت قبولاً واسعاً.



الشكل ٢ ـ ٣. طبولوجيا لغات الجيل الثاني والثالث.

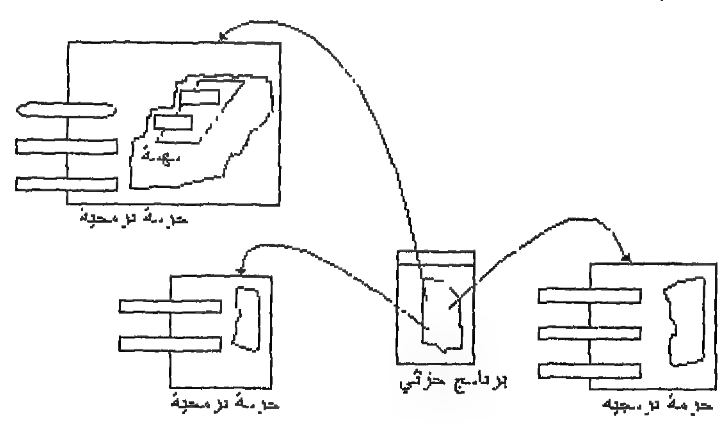
وبما أنّ لغات المجمع، تمثّل اللغات المستخدمة عموماً في النظم المحمولة، فقد قدمنا توبولوجيا هذه اللغات في الشكل ٢ ـ ٤. وتبدو هذه الرسمة مجردة قليلاً، لكنها توضّح بأنّ لغات المجمع لا تقدم أي بنية متلاحمة. وكل بنية موجودة في هكذا نظام، تكون مفروضةً من قبل الإدارة والتدريب، بين أعضاء الفريق. وأنّ الأنظمة الضخمة باستخدام لغات المجمع، طبعاً، هي نتيجة معاناة وخبرة لهؤلاء المطورين، ويقدم هذا مرونةً كبيرةً لخلق النظم، ومن الممكن كتابة ترميز مجمعٍ بنيوي. وعلى أية حال، عندما يصل الحل إلى حجم معتدل، فإنّ طبيعة لغة المجمع تزيد من تعقيده.



الشكل ٢ ـ ٤. طبولوجيا لغات المجمع.

ولقد تمّ تطوير لغة ADA في نهاية فجوة الأجيال للغات. وهكذا فقد تأثرت بالطرق البرمجية المعاصرة. وبمعنى، فإنها أول لغات الجيل الجديد. الشكل ٢ ـ ٥ يمثل تبولوجيا ADA. وتوبولوجيا ADA ليست مستويةً، مثل تبولوجيا لغات الأجيال السابقة، لكنها بالأحرى، ثلاثية الأبعاد. (سنشرح معنى الرموز التي بالشكل، في الفصل ٤). ومع ADA، يمكننا وصف بنية نظامية لخوارزمياتنا أيضاً بشكل جيد مثل بُنى معطياتنا. وأيضاً، يمكننا التحكم بتعقيد الحلول، وذلك بالإخفاء الفيزيائي للتفاصيل غير الضرورية، في كل مستوى. وأكثر من ذلك، تساعدنا هذه التبولوجيا بتمركز قرارات التصميم. وهكذا، تكون صيانة بنية التصميم الأصلي أكثر سهولةً، حسب التعديلات المطروحة.

وفي الفصول التالية، سنفحص علاوة على ذلك، كيف تقدم ADA الميزات التي تدعم مباشرةً مبادئ هندسة البرمجيات، وبذلك، تساعدنا على تطوير حلولٍ برمجيةٍ قابلةٍ للتعديل، وفعالة، وموثوقة، وقابلةٍ للفهم. وفي الفصل التالي، نقدم طريقة تطويرٍ غرضية التوجه، تنفذ هذه المبادئ، وبالإضافة لذلك، تستخدم إمكانية تبولوجيا ADA.



الشكل ٢ ـ ٥. طوبولوجيا Ada.





3

التصميم غرضي التوجه
Object-Oriented Design

حدود الطرق الوظيفية
طريقة التصميم غرضية التوجه
«آدا» ADA، كلغة تصميم



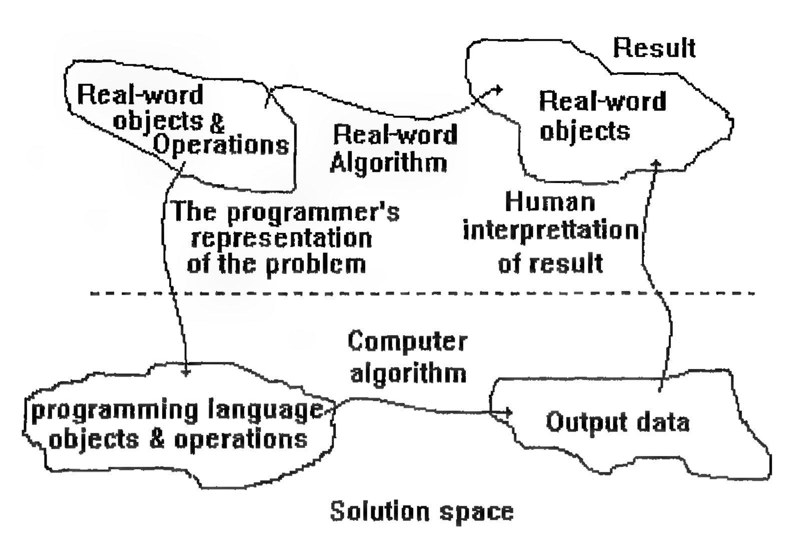
فحصنا في الفصل السابق، أهداف هندسة البرمجيات، والمبادئ الأساسية التي تساعدنا في الحصول عليها. وطوال عرضنا، كان الهدف واضحاً: يجب أن نطبِّق طرقاً، ولغاتٍ، ومحيطاتٍ مناسبة، لإدارة تعقيد النظم البرمجية. وفي هذا الفصل، سنقدم طريقة تصميم – وهي طريقة موجهة لخلق نظم برمجية معقدة. وبعد ذلك، سوف نرى كيف تدعم وتستخدم طريقتنا ميزات ADA.

٣ ـ ١ ـ حدود الطرق الوظيفية

(Limitations of Functional Methods):

ومهما كان التطبيق، فإنّ فضاء مسألته يترسخ في مكان ما من العالم الحقيقي، ويتنفذ فضاء حله، بإجراء توافق بين البرمجيات، والبنية الصلبة. ولقد اقترح H.Ledgard، نموذجاً لوصف مهمةٍ برمجيةٍ نموذجية – النموذج المبين في الشكل ٣ ـ ١، ويوضح هذا الشكل، أنّه في فضاء المسألة، يكون لدينا مجموعة أغراض من العالم الحقيقي، كل واحد منها، يملك مجموعةً من العمليات الموافقة. ويمكن لهذه الأغراض، أن تكون أبسط من دفتر حساب، أو أعقد من سفينة فضاء. ولدينا في فضاء المسألة، أيضاً، بعض خوارزميات العالم الحقيقي، التي تعمل على هذه الأغراض، وتقدم أغراضاً محولةً كنتائج. فعلى سبيل المثال، يمكن لنتيجة عالمٍ حقيقي، أن تكون دفتر حساب متوازن أو تغير مسار سفينة فضائية.

وعندما نطور نظاماً برمجياً، إما أن ننمذج مسألةً من العالم الحقيقي، أو كما في حالة النظم المحمولة، نأخذ أغراضا من العالم الحقيقي، ونحولها إلى وحداتٍ برمجية، وبنيةٍ صلبة، لإنتاج نتائج العالم الحقيقي. ومهما كان التنفيذ، يجب أن يكون فضاء حلنا، موازياً لفضاء مسألتنا. فأولاً، تقدم لغات البرمجة، التقنيات التي تسمح للمبرمج بتمثيل أغراض العالم الحقيقي؛ وبشكلٍ أساسي، يُجرِّد المبرمج الأغراض في فضاء المسألة، وينفذ هذا التجريد في البرنامج. وفيما بعد،



الشكل ٣ ـ ١. موديل لمهمة برمجية نموذجية

يتم تطبيق خوارزميات الحاسوب، التي تحوّل هذه الأغراض البرمجية. ومن ثـم، يجب على المبرمج استخدام تجريده المنطقي للعمليات، في العـالم الحقيقـي. وأخـيراً، تنتج هذه الخوارزميات شكلاً ما من الخرج، الذي يمكن أن يوافـق مباشـرةً، أعمـالاً مـا من العالم الحقيقي، مثل حركة جنيحة، أو يفسر من قبل الناس، في زمن غير حقيقي.

ومن الواضح، أنّه كلما وافق فضاء حلنا، لتجريد فضـاء مسـألتنا، كلمـا كـان مـن الأفضل الحصول على أهدافنا، من التعديل، والفاعلية، والوثوقية، وقابليـة الفهـم. وإذا كانت حلولنا بعيـدة عـن فضـاء المسـألة، يجـب أن نُجـري تحويـلاً ذهنيـاً أو فيزيائيـاً لتجريدات العالم الحقيقي، وهذا يزيد بشكل بديهي، قابلية تعقيد حلنا.

وإذا فحصنا اللغات الإنسانية، نجد أنّ جميعها تملك مركبتين أساسيتين، الجمل الإسمية، والجمل الفعلية. وتوجد بنية موازيـة فـي لغـات البرمجـة، لأنهـا تقدّم بُنـى لتنفيذ الأغراض (جملاً إسمية)، وعملياتٍ (جملاً فعلية). بينما، إنمعظم اللغات المطـورة قبل ظهور ADA هي بشكلٍ أساسيٍ، أمرية، وهذا يعنـي، أنهـا تقدم مجموعـةً غنيـةً

بالبنى لتنفيذ العمليات، لكنها بشكل عام، ضعيفة فيما يخص تجريد أغراض العالم الحقيقي. وأكثر من ذلك، مثلما رأينا، تشير تبولوجيا هذه اللغات، بأنّ معظم اللغات، لها بنية مسطحة نسبياً. وفي حين أن العالم الحقيقي ليس مسطحاً، وليس تسلسلياً، لكنه على العكس، متعدد الأبعاد، وموازي بشكلٍ صريح. وبالتالي، فإن الأجيال الثلاثة الأولى من لغات البرمجة، وخصوصاً لغات المجمعٍ، وسعت الفجوة بين فضاء المسألة، وفضاء الحل.

ولقد ناقشنا سابقاً، ثلاث طرق تصميم:

- التصميم البنيوي (التنازلي)، من الأعلى للأدنى.
- تصميم بنية المعطيات.
- التصميم غرضي التوجه.

فالتصاميم التنازلية من الأعلى للأدنى، ذات طبيعة أمرية ـ وهذا يعني، أنها تجبرنا على التركيز على العمليات في فضاء الحل، مع نظرةٍ صغيرةٍ لتصميم بنى المعطيات. والنظم التي صُممت بطرق كهذه، تتجه لعرض تبولوجيا، مشابهة لتبولوجيا لغات تقنيات الجيل الأول، والجيل الثاني. ونتيجة لذلك، فإن المعطيات هي بالضرورة عامة. أضف إلى ذلك، جعل النظام أقل وثوقية، بسبب إمكانية الإتصالات الخاطئة بين المسارات. والنظام أيضاً أقل مرونة، لأنّ تغير المعطيات يتجه نحو الإنتشار، خلال البنية التامة بكاملها .ويكون تصميم بنية المعطيات، على الجهة الأخرى من الطيف؛ وتركز هذه الطريقة على الأغراض، وتعالج العمليات بأسلوبٍ عام.

فباستخدام طريقتي التصميم هاتين ، يمكننا من جهة، الحصول على حلٍ وظيفيٍ بشكلٍ كامل، وتجنب تنفيذٍ معقولٍ لتجريدنا لأغراض العالم الحقيقي. ويمكننا من جهةٍ أخرى، الحصول على بُنى معطياتٍ واضحةٍ تماماً، لكن العمليات، هي التي ستكون مبهمة. وهذا يشبه قليلاً محاولة تكلم اللغة الإنكليزية، باستخدام الأفعال فقط، أو الأسماء فقط. وعلى الأقل، سنجري تحويلاً ذهنياً من فضاء الحل، لفضاء المسألة. وفي أسوأ حالة، سنجري تحويلاً فيزيائياً. وفي أية حال، تقدم لنا هذه الطرق، حلاً بعيداً جداً عن فضاء المسألة.

٣ ـ ٢ ـ طريقة التصميم غرضية التوجه

(An Object-Oriented Design Method) :

إن ما نرغبه، إذاً، هو طريقة تدعنا نوفّق مباشرةً تجريدنا للعالم الحقيقي، إلى بنية حلولنا. وبالإضافة لذلك، كما في اللغات الإنسانية، نبحث عن معالجةٍ موزعةٍ بين الأغراض والعمليات، في نموذجنا الحقيقي. وكثيراً ما ندعو هذا، بطريقة التصميم غرضية التوجه، ليؤكد الحقيقة بأنّه ليس أمرياً بشكلٍ صرف، وليس تصريحياً بشكلٍ صرف. وبالمقابل، تدرك هذه الطريقة أهمية الأغراض البرمجية كعوامل، كل واحد منها يملك مجموعة عملياته الخاصة به، والقابلة للتطبيق.

تختلف إذاً طريقة التصميم غرضية التوجه، بشكلٍ أساسيٍ، عن الطرق التقليدية، والتي من أجلها (الطرق التقليدية) يكون المعيار الأساسي لتحليل النظام، متمثلاً بأنّ كل وحدةٍ في النظام، تمثل خطوةً رئيسيةً للمعالجة العامة. ومع التقنيات غرضية التوجه، نطبق معياراً مختلفاً: تشير كل وحدةٍ في النظام لغرض، أو لصف أغراضٍ من فضاء المسألة. ويشكل كل من التجريد، وإخفاء المعلومات، القاعدة لكل تطويرٍ غرضي التوجه.

وبما أنّ رمز الغرض، يلعب دوراً مركزياً في هذه الطريقة، فمن الطبيعي أن نشرح ما نعنيه بالحد غرض. أولاً، إن الغرض يمثل كياناً له حالة، وهذا يعني، أن له قيمة ما. فعلى سبيل المثال، في نظام تحكم القيادة في سيارة، يمكننا معالجة الفرامل، والفالة (throttle)، ودواسة البنزين، والمحرك، كأغراضٍ معبرةٍ لنموذجنا الحقيقي. وأكثر من ذلك، فإن سلوك غرضٍ محددٍ، يُعّرف بالأعمال التي لحقت به، والعكس صحيح . وفي نفس نظام تحكم القيادة، تتضمن العمليات الهادفة على الفالة، زيادة أو نقصان قيمتها. ويجب أن تكون الفالة بدورها، قادرةً على التأثير في المحرك، وبالتالي، ضبط سرعته. وأخيراً، إن كل غرضٍ يكون بالفعل، نسخةً من صفٍ ما من الأغراض. فعلى سبيل المثال، إن نظام تحكم قيادةٍ محدد، يتفاعل مع محركٍ محددٍ ما. ويمثل هذا المحرك، نسخةً واحدةً من صف أغراض، التي هي جميعها، محركات. ويمكن أن يكون لدينا محركاً بست أسطوانات، أو ثمانية اسطوانات؛ وبرغم ذلك، فإن

جميع المحركات، تقدم السلوك ذاته. وإن الفائدة من وجهة النظر هذه، تتمثل بأنه بتجميع الأغراض في مجموعاتٍ ذات أغراضٍ مرتبطة، نحلل الصفات المشتركة إلى عوامل، ونمركز قرار تصميمنا لجميع نسخ الصف.

ويجب أن نتذكر دائماً، بأنّ التصميم غرضي التوجه، ليس إلا طريقةً جزئيةً لدورة الحياة؛ ويركّز على تصميم وتنفيذ مراحل التطوير البرمجي. (وسنناقش طبيعة دورة الحياة، بتفصيلٍ أكثر، في الفصل ١٩). ومثلما لاحظ Abbott " بالرغم من أن الخطوات التي نتبعها لتشكيل الإستراتيجية، يمكن أن تبدو آلية..... فإن الإستراتيجية تتطلب معرفةً عميقةً في العالم الحقيقي، وفهماً حدسياً للمسألة". وبالتالي، من الضروري ربط التصميم غرضي التوجه، مع متطلباتٍ موافقة، وطرق تحليلٍ من أجل المساعدة في خلق نموذجنا للحقيقة. ولقد وجدنا أنّ تطوير Jackson البنيوي (JSD)، سيوافق هذا بطريقةٍ واعدة. وإن تقنيات تدفق المعطيات العامة، مثل تقنيات Gane و Sarson، تمثل أيضاً، أدواتٍ مفيدةً في بناء تجريد الحقيقة. وإن المناقشة الكاملة لهذه التقنيات، خارجة عن نطاق هذا الموضوع، ولكن من أجل حاجاتنا، يكفي أن نعمل نموذجاً ذهنياً لتجريدنا للحقيقة. وإن شكلية JSD، أو طرق تدفق المعطيات، هي شيء أساسي لدعم الإتصال، الذي يجب أن يبقى حاضراً في النظم الضخمة، حيث يكون المطورون منفصلين زمنياً ومكانياً.

ومهما تكن الطريقة المختارة، يجب أن ندرك أنّه لا يمكننا التوقع أن نملك معرفةً تامةً لمجال المسألة. وعلى الأصح، إن نمو فهمنا دائماً تكراري. فكلما تعمقنا في تصميم حلّنا، كلما اكتشفنا سماتٍ جديدةٍ للمسألة، لم نكن نُدركها من قبل. وعلى أية حال، بما أنّ حلّنا يوافق المسألة مباشرة، فهذا الفهم الجديد لفضاء المسألة، لا يؤثر بشكلٍ جوهري على بنية حلّنا. ومع الطريقة غرضية التوجه، نكون عادةً قادرين على تقييد مدى التغيير لهذه الوحدات في فضاء الحل، والذي يوافق الأغراض في فضاء المسألة.

ويمكننا تخيل برنامجٍ ينفذ وحدة الحقيقة، كمجموعة أغراضٍ تتفاعل فيما بينها. ويمكننا تصميم نظامٍ باستخدام طريقةٍ غرضية التوجه، بإتباع الخطوات التالية:

- تحديد الأغراض وتسمياتها.
- تحديد العمليات التي تؤثر على كل غرض، والعمليات التي يجب أن تُحرَّض من قبل كل غرض.
- تأسيس رؤية كل غرضٍ مع بقية الأغراض.
- تأسيس واجهة تخاطبٍ لكل غرض.
- تقييم الأغراض.
- زرع كل غرض.

لقد استنتجنا هذه الخطوات، من الطريقة التي وضعها Abbott.

تحديد الأغراض (Identify The Objects) :

إن الخطوة الأولى لتحديد الأغراض وخصائصها، تتضمن إدراك الأفعال الأساسية، تجاه مخدّمات فضاء المسألة، بالإضافة إلى دورها في نموذجنا للحقيقة. وعادةً، تشتق الأغراض التي نحددها في هذه الخطوة، من الأسماء التي نستخدمها في وصف فضاء المسألة. أيضاً، يمكننا وصف عدة أغراض هامة، من طبيعةٍ مشابهة. وفي هكذا حالة، يجب أن نبني صفاً من الأغراض، حيث سيكون هنالك عدة نسخ. فعلى سبيل المثال، لاحظ Abbott، أنّه في وصف نموذجنا للحقيقة، يمكننا اكتشاف عدة أنواعٍ من الجمل الإسمية:

- تدعى الأسماء المشتركة، صف كيانات (مثل: الطاولة، الطرفي، الحساس).
- وتدعى أسماء الكميات، ووحدات القياس، النوعية، النشاط، المادة، أو كمية من نفس الكيان (مثل: الماء، المادة، الوقود).
- وتدعى الأسماء الصرفة، والأسماء التي من مرجعٍ مباشر، نسخاً معينة (مثل: حساس الحرارة، طاولتي، مفتاح الإطفاء).

ولا يمكن للأسماء المشتركة، وأسماء الكميات، وأسماء وحدات القياس، تحديد نسخٍ معينة، لكن في الواقع، تخدم في تحديد صفٍ من الأغراض، الذي يمكن أن نميزه كأنواع معطياتٍ مجردة (لاحظ الفصل ٦).

تحديد العمليات (Identify the Operations) :

تُستخدم خطوة التحديد، لتوصيف سلوك كل غرض أو صف أغراض. وقد بنينا هنا دلالة الغرض، بتحديد العمليات التي يمكن إنجازها بشكل أساسيٍ على الأغراض، أو بواسطة الأغراض. ونحن أيضاً حتى هذا الوقت، نبني السلوك الديناميكي لكل غرض، بتحديد القيود الزمنية، والحجمية، التي يجب أن تحترم. فعلى سبيل المثال، يمكننا تحديد ترتيبٍ مؤقتٍ للعمليات، مثل إنجاز "فتح" قبل إنجاز "إغلاق".

تأسيس الرؤية (Establish the Visibility) :

عندما نبني رؤية كل غرضٍ مع بقية الأغراض، نحدد الإرتباطات الساكنة بين أغراض، وصفوفٍ من أغراض – بمعنى آخر، ماذا "ترى" الأغراض، وما هو الغرض الذي "تراه". وإن الهدف من هذه الخطوة، هو التقاط تبولوجيا أغراضٍ من نموذجنا للحقيقة. وفي الفصل التالي، سنتعلّم بعض الرموز، التي يمكننا تطبيقها لوصف علاقات الرؤية والمدى.

تأسيس واجهة التخاطب (Establish the Interface) :

لبناء واجهة تخاطبٍ لكل غرض، نُنتِج توصيف وحدة، باستخدام رمزٍ صوريٍ مناسب (في حالتنا ADA). وتخدم هذه الخطوة، بالتقاط الدلالات الساكنة لكل غرض، أو صف أغراض، التي بنيناها في الخطوة السابقة. ويخدم هذا التوصيف أيضاً، كعقدٍ بين "زبائن" غرض، والغرض نفسه. وبمعنى آخر، تشكل واجهة التخاطب، الحدود بين الرؤية الخارجية، والرؤية الداخلية للغرض.

تقييم الأغراض (Evaluate the Objects) :

إن تحديد سمات فضاء المسألة التي ستصبح أغراضاً لفضاء الحل، لا تعتبر مهمة سهلة. حتى أنّه لا يمكن أن يتوقع المصممون الأكثر خبرة دائماً إيجاد الأغراض "الصحيحة"، من المحاولة الأولى. وبالتالي، فإن تقييم الأغراض (والتصميم ككل)، يكون خطوةً متعارضةً في التصميم، لتحديد إذا ما كان يجب عليك التكرار، لتحديد أغراضٍ جديدة. وفي الفصل ١٠، سنقدم عدداً من المساعدات المستخدمة لتقييم الأغراض.

زرع كل غرض (Implement Each Object) :

تستلزم الخطوة السادسة والأخيرة، زرع كل غرض واختيار تمثيلٍ مناسبٍ لكل غرض، أو صف أغـراض، وتنفيذ واجهـة التخـاطب مـن الخطـوة السـابقة. ويمكـن أن يتضمن هذا التركيب، أو إجراء تحليلات جديدة، أو كليهما. وأحياناً، سـيتكوّن غـرض ما، من عدة أغراض ثانوية. ففي هذه الحالة، يمكننا تكرار طريقتنا، لإجراء تحليلٍ أكثر للغرض. وفي أحيانٍ كثيرة، يتم زرع غرضٍ بالتركيب، بالاعتماد على أغراضٍ أو صفوف أغراضٍ موجودة، ذات مستويات منخفضة. وعند نمذجة نظام، يمكن أن يختـار المطـور تأجيلَ زرع جميع الأغراض، حتى الأخير. في هذه الحالة، يعتمد المطوِّر على توصيـف الأغراض، من أجل تجريـب بنيـة وسـلوك النظـام. وبشـكل مشـابه، يمكـن أن يحـاول المطوِّر عدة تمثيلات اختيارية خلال حياة الغرض، لتجريب سلوك عدة تنفيذات.

وسوف لا نقدم مثالاً كاملاً حتى الفصل الخامس، وسندرس الأدوات الضرورية المقدمة من قِبَل لغة تنفيذنا. وعلى أية حـال، وحتـى هـذه اللحظـة، يمكننـا رؤيـة كيفية دعم طرقنا لمبـادئ هندسـة البرمجيـات. ومـن الواضـح، أن طريقـة التصميم غرضية التوجه، تدعم التجريد وإخفاء المعلومات، بمـا أنّ تأسـيس هـذه الطريقـة، يتمثل بالإسقاط المباشر لنموذج الحقيقة، داخل فضاء الحل. وبالإضافة لذلك، فمـع ADA كلغة تصميم، يمكننا أن نخفي فيزيائياً، تفاصيل عملياتنا بشكلٍ جيـد، كمـا هو الحال في توصيف الأغراض.

وتقدم هذه الطريقة أيضـاً إسـتراتيجيةً هادفـةً لتحليـل نظـامٍ إلـى وحـدات. وباستخدام هذه الإستراتيجية، نمركز قرارات تصميمنا بطريقة، توافق رؤيتنـا للعـالم الحقيقي. وأكثر من ذلك، فلدينا حالياً رمز منتظـم لاختيـار الأغـراض والعمليـات، التي تمثل جزءاً من تصميمنا. وبالطبع، مثلما هو الحال مع أيّة إستراتيجية تطوير، يجب أن نعتمـد علـى بعـض أدوات الإدارة، لتـأكيد تكامليـة وصلاحيـة تصميمنـا. ولكن، بسبب البنية المفروضة التي تقدمها الطريقة غرضية التوجه، تصبـح مهمتنـا أكثر سهولة.

٣ ـ ٣ ـ ADA كلغة تصميم (ADA as a Design Language) :

أيّة لغة، إنسانية أو حاسوبية، تنجز شيئين للمستخدم: أولاً، تقدم مجالاً من التعبير. فعلى سبيل المثال، لا تملك لهجة أسكيمو أكثر من ٣٠ كلمة للثلج. وبشكل مشابه، فإنه مباح لمبرمجٍ بلغة APL أن يفكر بدلالة الأشعة، بسبب إغناء اللغة بمجموعة عملياتٍ على الأشعة. وثانياً، تقيِّد اللغة تفكير المستخدم. فعلى سبيل المثال، لاحظ إغناء اللغة الإنكليزية بمفرداتٍ تعبر عن أفعال، بينما العديد من اللغات الأوربية غنية بمجموعة أسماء؛ ولنفكر أيضاً بمبرمجٍ بلغة FORTRAN، تطلب منه حل مسألة باستخدام العودية.

وضمن أيّة لغة برمجةٍ محددة، يجب أن تقدم أدوات كافية، لتجيز لنا التعبير عن حل مسألة. وبشكلٍ مثالي، نرغب باستخدام لغةٍ، تجعلنا نعكس مباشرةً رؤيتنا لفضاء المسألة. ومع لغاتٍ من الأجيال المبكرة، غالباً ما نلائم الحل للغة، بدلاً من أن نكيف اللغة للحل. وكنتيجةٍ لذلك، فإن أدواتنا تتعارض مع الهدف الأساسي، الذي هو حل المسألة. وفي حالة برمجة فضاء مسألةٍ بلغة ما، تعكس مباشرةً بنيتنا لفضاء المسألة، نحصل على تنفيذٍ قابلٍ للفهم، ويسهِّل إدارة تعقيد النظم الضخمة. ويجب أن تقدم لغة من هذا النوع، أدواتٍ للتعبير عن أغراضٍ وعملياتٍ أساسية، وأن تكون، أكثر من ذلك، قابلةً للتوسع، من أجل أن تسمح لنا ببناء أغراضنا، والعمليات التجريدية الخاصة بنا، وحتى يكون الوضع كاملاً يجب أن تحترم لغتنا هذا التجريد.

و ADA تجيب على هذه المتطلبات. فهي ليست فقط مناسبةً كلغة زرع، لكنها معبرة بشكلٍ كافٍ، لتُستخدم كوسيلةٍ لالتقاط قرارات تصميمنا. وتقدم مجموعةً غنيةً من البنى، لوصف أغراضٍ وعملياتٍ أساسية. وبالإضافة لذلك، تقدم بنية حزمةٍ، يمكننا معها بناء تجريداتنا الخاصة. وفي الفصل التالي، سنبدأ بفحصٍ معمقٍ للتفصيلات التقنية للغة، باستخدامها مقترنةً مع طريقتنا، في التصميم غرضي التوجه. ولاحظ بأنّ هذه الطريقة، هي إحدى النماذج التي بواسطتها نطبق ADA؛ كما ويمكننا استخدامها، مع أية طريقة تصميمٍ تقليديةٍ أخرى. وعلى أية حال،

تقدم ADA للمطور بعض الميزات الفريدة، غير الموجودة في معظم اللغات، بما في ذلك، مهمة معالجة الإستثناء والحزم البرمجية. وكنتيجةٍ لذلك، تتطلب إمكانيات ADA، أن نوقف أنفسنا بعيدين عن التفكير المسطح والتسلسلي، الذي تجبرنا به بقية اللغات، وتبدل تفكيرنا بدلالة حدود فضاء المسألة. وإن طريقة التصميم غرضية التوجه، تقدم هكذا نموذج.





لمحة عن اللغة

An Overview Of the Language

متطلبات اللغة.

«آدا» ADA من الأعلى للأدنى.

«آدا» ADA من الأدنى للأعلى.

ملخص عن ميزات اللغة.



بعد عرض المبادئ الأساسية لهندسة البرمجيات العصرية، نحن الآن جاهزون لفحص لغة ADA نفسها. والهدف من هذه المقدمة الظريفة، يتمثل في إعطاء لمحةٍ عن قوة ADA، ونعرض في الوقت ذاته، جزءاً من بنيتها ومصطلحاتها. فلا تبحث في فهم كل شيءٍ بشكلٍ عميق حتى الآن: وهناك أمثلة كثيرة تمّ تقديمها في هذا الفصل، بهدف توضيح شكلٍ ADA. وأكثر من ذلك، لا تتوقع أن تفهم المضمون الكلي من كل مثال. فمناقشة دقيقة لكل بنية، ستأتي في فصول لاحقة. والآن، سنطبق هنا مبدأ إخفاء المعلومات، ولن ندرس إلا التجريد في أعلى مستوى .

٤ ـ ١ ـ متطلبات اللغة (Requirements for the Language):

آخذين بعين الإعتبار مجال المسألة التي تمّ تصميمها، فليس عجباً أن تكون ADA ذات قدرة تعبيرية هامة. وخاصةً إذا ما اعتبرنا دفتر الشروط STEELMAN، الذي يطلب أن تكون لغةً قادرةً على تقديم:

- التراكيب البنيوية.
- التنوع القوي.
- توصيف للدقة، نسبي ومطلق.
- إخفاء المعلومات، وتجريد المعطيات.
- المعالجة المتوازية.
- معالجة الإستثناءات.
- التعاريف المولّدة.
- تسهيلات متعلقة بالآلة.

وقد تجاوزت في بعض الأحيان هذه المتطلبات حالة الفن، لكنها سببت العديد من تصاميم لغات البرمجة، لتجتمع في لغةٍ برمجةٍ عالية المستوى، ووحيدة. ومن وجهة النظر هذه، فإن ADA ليست لغةً جديدة؛ وفي الواقع، إنها تمثل تضافراً متماسكاً لجهدٍ هندسي.

فخلال خلق أي نظام، يجب أن يُجري المطورون توازناً بالتنفيذ؛ وهذا ما طُبق على ADA. ولذلك، فقد أسس فريق ADA ثلاثة أهدافٍ للغة، لقيادة قرارات تصاميمهم، هي:

- التعرف على أهمية قابلية الصيانة، ووثوقية البرنامج.
- اعتبار البرمجة نشاطاً إنسانياً.
- الفاعلية.

هذا، وقد تم بشكل خاص، اعتبار أهداف الوثوقية، وقابلية الصيانة (قابلية التعديل)، الأولى من بين الأهداف. وبالفعل، مثلما لاحظنا في الفصل ١، فهي التي تستهلك الموارد في دورة حياة البرمجي. أما النشاط الإنساني البرمجي، فقد تمّ اعتباره لاحقاً، مع التأكيد على ما يساعد المطورين على إدارة تعقيد الحلول البرمجية. ومثلما سنرى، فبرامج ADA المبنية بشكلٍ جيد، تكون سهلة القراءة (بالرغم من أنه في بعض الأحيان، تكون كتابتها مملة). وفي الواقع، هذا حل وسط معقول، باعتبار أنّ المبرمج لن يكتب قطعة ترميزٍ، إلا مرةً واحدة، ولكن ستتم قراءة هذا الترميز، العديد من المرات. هذا، وإن قواعد اللغة تملك طرقاً عديدة لقيادة المبرمج نحو استخدام تصميم وأسلوب ترميز جيدين. وأكثر من ذلك، إن استخدام بيئة دعم برمجة ADA يُتمّم اللغة، ويمكن أن يجعل تطوير نظم ADA أسهل. أما في الهدف الثالث، فقد اعتبر فريق تصميم اللغة فعالية التنفيذ والزرع. وقد تمّ نبذ كل بنية لغة، يكون زرعها غير واضح، أو يتطلب موارد آلة زائدة.

٤ ـ ٢ ـ ADA من الأعلى للأدنى

(ADA from the Top Down):

يكون برنامج ADA (كما نفضل استدعاءه، نظام ADA) مكوناً من وحدةٍ برمجيةٍ أو أكثر، وكل وحدةٍ برمجية، يمكن أن تترجم بشكل منفصل. وتتألف الوحدات البرمجية من برامج جزئية، ومهام، وحزم برمجية، ووحدات مولّدة. والبرنامج الجزئي، إما أن يكون إجرائيةً، أو تابعاً فرعياً.

ويعبر عن سلسلة من الأفعال. من جهة أخرى، تعرف المهمة فعلا يتم تنفيذه على التوازي مع مهام أخرى. يمكن زرع مهمة على معالج وحيد، عدة معالجات، أو شبكة حاسبات.

تمثل الحزمة البرمجية تجمعاً لموارد تقديرية، يمكن أن تغلب أنـواع معطيـات، أغراض معطيات، برامج جزئية، مهام، أو حتى حزم برمجيـة أخـرى. يتمثـل الهـدف الرئيسي من الحزمة البرمجيـة بـالتعبير وإجبـار تجريـدات المسـتخدم المنطقيـة داخـل اللغة. أخيراً، الوحدة المولّدة تمثل قالباً للبرامج الجزئية والحزم البرمجية وتقـدم الآليـة الأساسية لبناء مركبات برمجية يمكن إعادة استخدامها.

يلخص الجـدول ١ مـيزات كـل وحـدة برمجيـة فـي ADA وبالإضافـة لذلـك، يجدول تطبيقات كل وحدة، وسندرس هذه التطبيقات بالتفصيل في فصول لاحقة.

الجدول ١

الوحدة البرمجية	الميزات	التطبيقات
برنامج جزئي	أفعال تسلسلية	وحدات البرامج الرئيسية
		تعريف تحكم وظيفي
		تعريف عمليات على الأنواع
حزمة برمجية	مجموعة من الموارد	تسمية مجموعة تصريحات
		تجميع وحدات برمجية مترابطة
		تجريد أنواع معطيات
		تجريد حالة الآلة
مهمة	أفعال متوازية	أعمال متنافسة
		تدوير الرسائل
		التحكم بالموارد
		المقاطعات
وحدة مولّدة	قالب	إعادة استخدام المركبات البرمجية

وبشكلٍ عام، فإن جميع الوحدات البرمجية بـ ADA، لها بنيـة وحيـدة مؤلفـة من جزئين ، وهما التوصيف، والجسم. ويعّرف التوصيف المعلومات المرئية لـلزبون مـن وحدة البرنـامج (واجهـة التخـاطب)، بينمـا يحتـوي الجسـم علـى التفـاصيل المزروعـة للوحدة، والذي يمكن أن يكون منطقياً ونصياً مخفياً علـى الزبـون. وللمسـاعدة فـي إدارة

تطوير الحلول الضخمة، يمكن ترجمة كلاً من الجسم والتوصيف بشكل منفصل عن بعضهما البعض. وبالتالي، يمكن للمطور أن يكتب توصيف الوحدات البرمجية من أعلى مستوى، وبالتالي، خلق بنية تصميمٍ للحل موثوقة. وفي نهاية معالجة التطوير، يمكن للمطورين إضافة أجسام الوحدات، ومن ثمّ يتم تنقيحهما بشكلٍ مستقل، ليكمل زرع النظام.

وبما أنّ مجال اعتماد ADA كان معقداً، يمكن لنظام ADA أن يتكون من مئات، إن لم يكن من آلاف الوحدات البرمجية. وكلما تضخم حجم حلّنا ، كلما ازدادت صعوبة فهمنا لبنيته. وبفضل الترميز الذي قدمناه في الفصل٢ ، يمكننا موافقة كل وحدة برمجية في ADA لرمزٍ وحيد. لاحظ كيف عبّرنا عن هذه الرموز، حتى نسمح بإظهار توصيف وجسم كل وحدة.

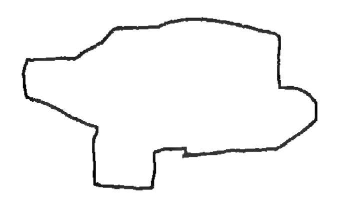
ومثلما يشير الشكل ٤ ـ ١ ، يمكننا تمثيل كيانٍ غير معرفٍ أو مخفي (الذي يمثل غرضاً)، على شكل نقطةٍ غير منظمة. ويرمز هذا الشكل، إلى أنّ بنية هذا الكيان لا تخصنا في هذا المستوى، وبالفعل، حتى أنها ليست مرئية. وفي الشكل ٤ ـ ٢ ، تمّ تمثيل البرنامج الجزئي كبنية خطية، متضمنين طبيعته التسلسلية. ونبرز هنا جزئيّ البنية للوحدة، بتميز توصيفها عن جسمها. ويبين الشكل ٤ ـ ٣، بأنّه يمكننا رؤية المهمة كمتوازي أضلاع، متضمنين طبيعته المتوازية. وكما في البرامج الجزئية، فإن تمثيل المهمات، تفصل التوصيف عن الجسم.

والحزم البرمجية، هي بنى في ADA هامة جداً، ومتعددة الإستعمالات، وهي تتطلب رمزاً خاصاً. ولقد وصف Ichbiah حزمةً برمجيةً، بأنها تشبه جداراً يحيط بمجموعة كيانات مرتبطة منطقياً. ومثلما يبدو في الشكل ٤ ـ ٤ ، نتصور جزءاً مرئياً (التوصيف) من حزمةٍ برمجيةٍ كنوافذ في الجدار. فإذا دورنا الجدار بزاوية مقدارها ٩٠ درجة على المحور Y ، نحصل على التمثيل البياني لحزمة برمجية، من وجهة نظر خارجية. ومثلما يبدو في الشكل٤ ـ ٥ ، نرى الآن النوافذ كعتبة نافذة. ونشير للجسم بشكل منفصل.

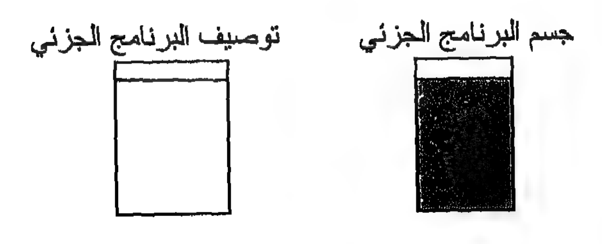
بما أنّ الوحدات المولّدة تمثل قالباً، فمن أجلها يمكننا تبني رمزي الحزمة البرمجية، والبرامج الجزئية. والشكل٤ ـ ٦ ، يوضح أن الوحدات المولّدة، تملك أيضاً بنيةً ذات جزئين.

ويمكننا بسهولة، تمثيل بنيةٍ برمجيةٍ معقدة، باستخدام هذه الرموز. فعلى سبيل المثال، يبين الشكل٤ ـ ٧، اتصال مهمتين. ويشير الخط، إلى أنّ المهمتين متكلتين على بعضها بعضاً؛ ويؤشر السهم من المهمة الداعية، إلى المهمة التابعة (المدعوة). وبما أنّه لا يمكن ترجمة المهمة بشكل منفصل (تفاصيل أكثر في الفصل ١٤)، فيجب تعشيشها في جسم حزمةٍ برمجية، ومثلما نرى في الشكل ٤ ـ ٨. يوضح الشكل٤ ـ ٩، أنه بتراجعنا للخلف لمستوى واحد، يمكننا رؤية البنية بشكلٍ عام. وفي هذا المثال، تشير الأسهم، بأنّ جسم البرنامج الجزئي، يستخدم خدمات الحزمة البرمجية (توصيفاتها)، وأيضاً توصيف حزمتين برمجيتين مولّدتين. لاحظ بأنّ تبولوجيا نظم ADA، ليست هرمية بشكلٍ صارم.

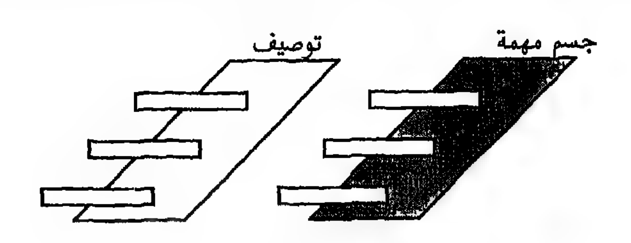
وأكثر من ذلك، نميل إلى استخدام الحزم البرمجية، والوحدات المولّدة كوحدات التحليل الأساسية.



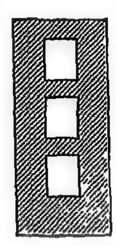
الشكل ٤ ـ ١. رمز لكيان غير معرّف.



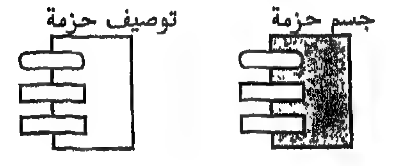
الشكل ٤ ـ ٢. رموز البرامج الجزئية في ADA.



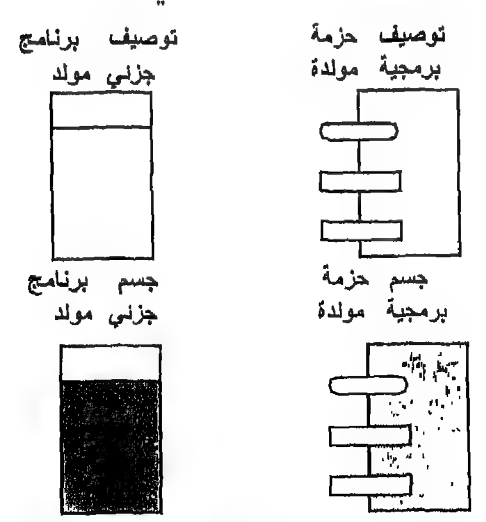
الشكل ٤ ـ ٣. رمز لمهمة في ADA.



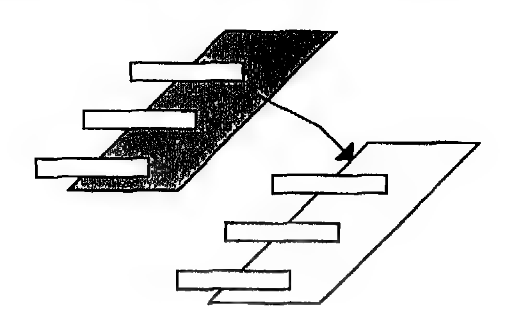
الشكل ٤ ـ ٤. حزمة برمجية مع جزء مرئي.



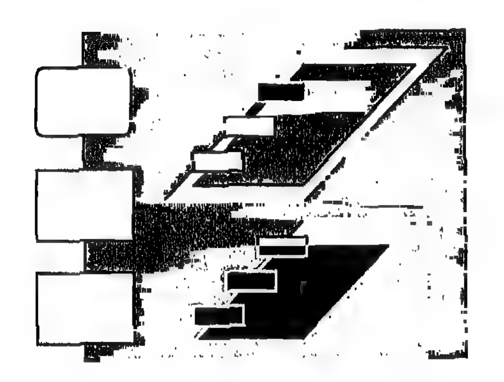
الشكل ٤ ـ ٥. حزمة برمجية في ADA.



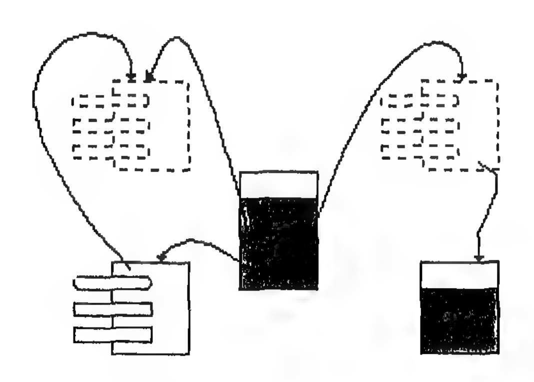
الشكل ٤ ـ ٦. اتصال المهام في ADA.



الشكل ٤ ـ ٧. اتصال المهام في ADA.



الشكل ٤ ـ ٨. وحدات برمجية متعششة في ADA.



الشكل ٤ ـ ٩. نظام ADA من الأعلى للأدنى.

٤ ـ ٣ ـ ADA من الأدنى للأعلى (ADA from the Bottom Up):

يمكننا كتابة كل بنية في ADA، مستخدمين مجموعة المحارف البيانية الأساسية، متضمنةً مايلي:

- *المحارف العلوية*: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z.
- *الأرقام العشرية*: 0 1 2 3 4 5 6 7 8 9.
- *محارف خاصة*: ."#'()*+,-./:;<=>_|&
- *المحرف الذي يمثل فراغ.*

وقد تطلبت STEELMAN هكذا مجموعة محارف، لتسهيل قابلية نقل وحدات البرنامج على مستوى محرف المنبع. وتسمح لنا ADA، بتوسيع مجموعة المحارف هذه، لتتضمن مجموعة محارف الـ ASCII البيانية، البالغ عددها ٩٥ محرفاً، متضمنةً:

- *المحارف السفلية*: *a b c d e f g h I j k l m n o p q r s t u v w x y z*
- *محارف خاصة*: *!$?@[\]`{}~^%*

ويمكن تحويل برنامجٍ مكتوبٍ بمجموعة المحارف الموسعة، إلى برنامجٍ مكافئ مكتوبٍ بمجموعة المحارف الأساسية.

وحدات المفردات (Lexical Units):

لقد تمّ بناء وحدات اللغة الأساسية، التي تدعى *وحدات المفردات*، من مجموعة محارف ADA . وتتألف وحدات المفردات من معرِّفات، ومعطياتٍ رقمية، ومعطياتٍ حرفية، وسلاسل محرفية، وحروف محددة، وتعليقات. وتتشكل *المعرِّفات*، من محرفٍ متبوعٍ بسلسلة محارف، وأرقام عشرية، و/أو الرمز _ (لزيادة قابلية القراءة). وأيضاً يمكن أن تكون المعرِّفات كلمات محجوزة؛ إذ أنّه يوجد في ADA / ٦٣ / كلمة محجوزة، كما هو مبين في الجدول ٢.

لا تفرض ADA حدوداً على طول أسماء المعرِّفات التي يحددها المستخدم، والتي يجب أن تتواجد على سطرٍ واحد. وفيما يلي، معرفات صحيحة في ADA:

- Colors_of_rainbow.
- temperature_sensor_37.
- Page_Count.
- POLL_TERMINALS.

الجدول ٢: كلمات ADA المحجوزة.

Abort	declare	generic	Of	select
Abs	delay	goto	Or	separate
Accept	delta		Others	subtype
Access	digits	if	Out	
All	do	in		task
And		is	Package	terminate
Array			Pragma	then
At	else		Private	type
	elsif	limited	Procedure	
	end	loop		
Begin	entry		Raise	use
Body	exception		Range	
	exit	mod	Record	when
			Rem	while
		new	Renames	with
Case	for	not	Return	
Constant	function	null	Reverse	xor

ولا تميز ADA بين الأسماء التي تستخدم محارف عالية أو سفلية، وبالتالي، تكون المعرّفات page_count, Page_Count, PAGE_COUNT متكافئة، لكن PAGECOUNT غير مكافئة، وبالتالي فإن الرمز _ يكون معبراً. وفي أمثلة لاحقة من هذا الكتاب، تكون معرّفات المستخدم مكتوبةً بمحارف مختلطة، كما هو الحال في Page_Count.

وتمثل *المعطيات الرقمية* قيماً حقيقيةً، أو صحيحةً دقيقة. ويمكن التعبير عن عددٍ في أية قاعدة، من القاعدة الثنائية، وحتى القاعدة الست عشرية. وبالإضافة لذلك، فقد تمّ السماح باستخدام الرمز _ بين الأرقام العشرية المشكّلة لعدد، ويساعد الرمز _ أيضاً بإغناء القراءة، إذ يتم في الواقع إهماله من قبل المترجم. وفيما يلي معطيات رقمية صحيحة:

```
7
1_000_000       -- the same as 1000000
1_00_00_00      -- the same value
1e6             -- the same value, read as 1*10^6
2#1100#         -- base 2 equivalent of 12_10
16#C#           -- base 16 equivelent of 12_10
```

وفيما يلي، معطيات رقمية حقيقية صحيحة:

```
0.125
3.141_592_65    -- the value of π
2.78e-3         -- equivalent to 0.00278
1.0e6           -- the same value as 1000000.0
16#F.0#         -- the same value as 1510
```

وتتطلب الأعداد الحقيقية على الأقل، رقماً واحداً من كل جانب من الفاصلة العشرية.

وتتألف *المعطيات الحرفية*، من أي محرفٍ من محارف ASCII البيانية الـ ٩٥، ويجب أن يكون المحرف بين أداتي حصر مفردتين. بينما تمثل *السلسلة المحرفية*، من جهة أخرى، سلسلة محارفٍ طولها صفر أو أكثر، محصورة بين أداتي حصر مزدوجتين. على سبيل المثال:

```
'A'               -- a character literal
'*'               -- another character literal
'''               -- a character literal whose value is"
""                -- a null string
"Time of day"     -- a string of length 11
""""""            -- a string whose value is "
```

تشكل *المحارف المحددة*، الصف الثاني من وحدات المفردات، وتتألف من رموزٍ بسيطة:

' ( ) * + , – . / : ; < = > | &

ويضاف لهذه المحارف المحددة، الرموز المركبة:

```
=>  ..  **  :=  /=  >=  <=  <<  >>  <>
```

ويوجد معنى خاص للمحارف المحددة، تتعلق بسياقها.

وأي عدد من الفراغات، يمكن أن يفصل بين وحدات مفردات متجاورة. ويجب أن توضع كل وحدةٍ مفردةٍ على سطرٍ واحد، ولكن يمكن أن تتواجد في أي مكان من السطر، وذلك لأنّ ADA، تمثل لغة برمجةٍ ذات حقول حرة. وتؤثر السطور أيضاً على ترتيب *التعليقات*، والصف الصوري الأخير من وحدات المفردات. ويبدأ التعليق بخطين قصيرين (—) وينتهي مع نهاية السطر.

وبالرغم من أنها لم تعتبر وحدة مفردات، فإنه يوجد صنف آخر من البنى، معرّف في أخفض مستوى من اللغة، وهي *العملياتي* ***Pragma***، من الكلمة اللاتينية ***Pragmaticus***

وهي موجهة فقط لمترجم ADA . وقد تمّ عرض وصفٍ لعملياتي ADA المسبقات التعريف، في الملحق B .

تعريف الأنواع، والتصريح عن الأغراض

(Type Definitions & Object Declarations):

من المعروف أنّ كل لغة برمجة، تتألف من:

- قواعد للتعبير عن اللغة نفسها.
- الأغراض، وآلية لتعريف الأغراض.
- العمليات، وآلية لبناء العمليات.

ففي المقطع السابق، وصفنا بعض هذه القواعد من أجل التعبير عن اللغة (وحداتها)، والآن نتجه إلى العنصر الثاني من اللائحة، والذي يتألف من أنواعٍ وأغراض. وتصف *أنواع المعطيات* المجردة:

- مجموعة قيم.
- مجموعة عمليات، يمكن تطبيقها على أغراض من النوع.

ففي الفصل السابق، تعلمنا صفوف الأغراض. ولأسباب تاريخية، فإن معظم لغات البرمجة – بما في ذلك ADA – قد استخدمت الحد *type* كتعبيرٍ مكافئ. فعلى سبيل المثال، يمكننا اعتبار Waiting_Line نوعاً. ومن الواضح أن سطر الإنتظار، يملك مجموعة قيم (الكيانات التي يمكنها أن تنتظر)، ومجموعة عمليات (إضافة إلى السطر، حذف من

السطر). لاحظ بأنّ نوعاً مثل Waiting_Line يعرف قالباً فقط. فإذا رجعنا إلى سطر الإنتظار الثالث، في مخزن البقال المجاور"، عندها نكون قد رجعنا لغرض من النوع Waiting_Line. ويمكن أن يشير هذا الغرض إلى قيمة خاصة، مثل "ثلاثة أشخاص في السطر."

وفي لغة ADA، هناك عدة صفوف للأنواع تكون صالحة، من هذه الصفوف:

- أنواع معطيات سلمية (Scalar data types):

  وتضم ما يلي:

  - الأنواع الصحيحة (Integer).
  - الأنواع الحقيقية (Real).
  - الأنواع المرقمة (Enumeration).
- أنواع معطيات الوصول (Access data types).
- أنواع معطيات خاصة (Private data types).
- أنواع جزئية وأنواع مشتقة (Subtype & derived types).
- أنواع معطيات مركبة (Composite data types):

  وتضم ما يلي:

  - المصفوفة (Array).
  - التسجيلة (Record).

ونعتبر صف الأنواع الأول (Integer, real, enumeration) أنواعاً سلمية، وذلك لأنها لا تملك أجزاء مركبة، وبالتالي، تعرف قيماً أحادية البعد. وتعرف *الأنواع الصحيحة* مجموعات من أعداد كاملة (أعداد دون جزء كسري). وتعرف *الأنواع الحقيقية* مجموعات أعداد مع جزء كسري (أعداد ممثلة بالفاصلة العائمة، أو بالفاصلة الثابتة). وتسمح *الأنواع المرقمة* للمستخدم، بتعريف مجموعة قيمٍ خاصة به. وفيما يلي أنواع صحيحة:

```
Integer                    -- a predefined type
Natural                    -- a predefined type, >= 0
type Index is range 1..50; -- a user defined type
```

وفيما يلي أنواع حقيقية:

```
Float                              -- a predefined type
type Mass is digits 10;            -- a floating-point type
type Voltage is delta 0.01         -- a fixed-point type
range -12.0..+24.0
```

وفيما يلي أنواع مرقمة:

```
Boolean                            -- a predefined type(False, True)
Character                          -- another predefined type
type Color is (Black, Red, Green,  -- a user-defined type
Blue, Cyan, Magenta, Orange,
White);
```

وإجمالاً، فإن الأنواع الصحيحة، والأنواع المرقمة تدعى **بالأنواع المتقطعة**.

وأيضاً تعرف ADA نوعين بنيويين، المصفوفات والتسجيلات. ***فالمصفوفة***، مؤلفة من مجموعة عناصر لها نفس النوع، بينما التسجيلة، مؤلفة من مجموعة عناصر، ليست بالضرورة من نفس النوع. فعلى سبيل المثال، نورد فيما يلي بعض الأمثلة الصحيحة، عن تعريف أنواع المصفوفات:

```
Type chess_board is array(1..8,1..8) of color;
        -- a two-dimensional array type
type pixel is array(Color) of float;
        -- an eight-element array type
type sensor is array(Index range 5..10) of Voltage;
        -- a six-element array type
type vector is array(positive range<>) of Integer;
        -- unconstrained array type
```

وفيما يلي بعض الأمثلة الصحيحة، عن تعريف أنواع التسجيلة:

```
type Date is record
        Day    : Integer range 1..31;
```

```
            Month : Integer range 1..12;
            Year    : Natural;
           end record;
type Value is record
            Name        : String(1..20);
            Location    : String(1..30);
            Open        : Boolean;
            Flow_Rate : Float range 0.0 .. 30.0;
            Inspected  : Date;
           end record;
type Value (Defined : Boolean := False) is
            record
             case Defined is
              when False => null;
              when True => Quantity : Integer;
             end case;
            end record;
```

وحتى الآن، لم نعرض أنواعاً إلا من أجل أغراضٍ ساكنة. وهذا يعني، تلك الأغراض التي تعرَّف في زمن الترجمة. وعلى أية حال، يمكن أن توجد حالات، يجب فيها خلق المعطيات ديناميكياً (خلال زمن التنفيذ). فعلى سبيل المثال، لنعتبر نظام تحصيل معطياتٍ، يغير معدل أخذ عيناته، بالإعتماد على قيم الدخل. ففي هذه الحالة، من غير الممكن التنبؤ عن حجم المعطيات التي يجب أن تُخزن. وفي ADA، تؤشر قيم الوصول على أغراضٍ أخرى، وتسمح لنا بخلق أغراضٍ ديناميكياً، والرجوع إليهم، بغرض وصول. على سبيل المثال:

```
type Buffer_Pointer is access Buffer;
              -- pointer to Buffer objects
```

ومثلما سنرى في الفصل ١٤، يمكننا تسمية مهامٍ كأنواع، مما يمكننا من خلق مهام جدد في زمن التنفيذ.

ومثل بقية صفوف أنواع المعطيات، تسمح ADA *بأنواعٍ خاصة*، تستخدم فقط، بالحزم البرمجية. وكمثل بقية الأنواع، تعرَّف الأنواع الخاصة، مجموعة قيم، ومجموعةً من العمليات الشائعة. ولكن بشكل مختلف عن بقية الأنواع، تكون بُنى الأنواع

الخاصة، غير مرئيةٍ للزبون. وأكثر من ذلك، يمكن للزارع أن يعرِّف بعض العمليات على الأنواع الخاصة، وتصبح هذه العمليات، هي الوحيدة التي يمكن أن يتعامل معها المستخدم. وتقدم الأنواع الخاصة آليةً، تجبر على إخفاء المعلومات، وخلق أنواع معطياتٍ مجردةٍ جديدة. فعلى سبيل المثال، فيما يلي نعرِّف مكدساً مجرداً:

```
package Stacks is -- package encapsulating a private type
 type Stack is private;
procedure Push (Element : in  Integer; On : in out Stack);
procedure Pop (Element:out Integer; From:in out Stack);
 private
Maximum_Elements : constant Integer := 100;
type List is array (1..Maximum_Elements) of Integer;
type Stack is
record
  Structure : List;
  Top       : Integer range 0..Maximum_Elements :=0;
end record;
end Stacks;
```

وبشكلٍ عام، يجب أن يملك كل توصيف حزمةٍ برمجيةٍ جسماً، ولكن من أجل هذا المثال، سنهمل زرع الجسم. والعمليات الوحيدة التي يمكن أن يطبقها الزبون على أغراض من نوع Stack، تكون Push , Pop. لأن قواعد اللغة، تمنع الزبون من الوصول إلى مركبات أغراض من Stack مباشرةً.

وأيضاً، تملك ADA، آليةً تسهِّل تحليل خواص أنواع الأصل، إلى عواملها الأولية، تُدعى *بالأنواع الجزئية والأنواع المشتقة*. وسنفحص في الفصل ٦ كيف يمكن لهذين الصفين من الأنواع، أن يدعما التجريد.

وتقدم أنواع المعطيات، آليةً فقط لوصف بنية المعطيات، بينما تخلق *التصريحات*، نسخاً مؤقتةً (أغراض) من نوعٍ محدد. وفي ADA، يسمح التصريح عن أغراض، بخلق متحولاتٍ وثوابت. فعلى سبيل المثال، فيما يلي تصريح عن متحولاتٍ صحيحة:

```
Counter    : Integer;          -- using a predefined type
Birth_Day  : Date;             -- using a user_defined type
My_Buffer  : Stacks.Stack;     -- another user-defined type
```

وفيما يلي، أمثلة صحيحة، عن أغراضٍ ثابتة:

```
Pi       : constant := 3.141_592_65;
Port_Address : constant Integer := 8#777_776#;
```

وتتميز ADA بأنها ذات تنويعٍ قوي، وهذا يعني أنه لا يمكننا مباشرةً تجميع أغراضٍ من أنواع مختلفة. وهذه حالة أخرى، من خلالها يمكن أن تجبر اللغة تجريد المستخدم.

الأسماء والتعابير (Names and Expressions):

في أيّة لغة برمجة، تستخدم الأسماء للإشارة إلى كياناتٍ مصرحٍ عنها. ففي مسائل ضخمة، يمكن أن يحتوي فضاء الأسماء، مئات إن لم يكن آلاف، من الأسماء. وبالنتيجة، قد يكون من الصعب لمبرمجٍ تجنب أسماء تمّ تعريفها مسبقاً، في الوقت الذي يحاول به، خلق أسماء ذات معنى، لكياناتٍ جديدة. وإن إحدى الآليات التي تستخدمها ADA لتجنب هذه المسألة، يتمثل *بالتحميل الزائد*. ويسمح التحميل الزائد للمبرمج، باستخدام نفس الإسم لكيانات مختلفة، شريطة ألا يسبب استخدام الإسم، التباساً. فعلى سبيل المثال، من المعقول إجراء تحميلٍ زائد على معطيات مرقمة:

```
type Sensor_Type is (Temperature, Humidity, Pressure);
        -- an enumeration type
type Alarm      is (Normal, Temperature, Intrusion);
        -- overloading the literal Temperature
```

وبالطبع، إن الإستخدام غير المقيد للتحميل الزائد، يمكن أن يشوش، ولذلك، يجب تطبيقه بحذر.

ومثلما سنناقش بالتفصيل في الفصل ٩، يمكن استخدام الأسماء، والثوابت، في التعابير لحساب قيمة. وتحتوي ADA على مجموعةٍ مألوفةٍ من العوامل، متضمنة (من أعلى أسبقية، لأدنى أسبقية):

```
**      not     Abs                          -- Heighest Precedence Operator
*       /       Mod     rem                  -- Multiplying Operator
+       -       &                            -- Binary adding Operator
=       /=      <       <=      >      >=    -- Relational operator
+       -                                    -- Unary adding Operator
and     or      xor                          -- Logical Operator
and     then    or      else                 -- short_circuit operator
```

باستخدام هذه العوامل، يمكن تشكيل عدة تعابير. فعلى سبيل المثال:

```
Pi                    -- a simple expression
(b**2)-(4.0*a*c)      -- a more complex expression
char in 'A'..'Z'      -- a Boolean expression
(2.789**4)+36.0       -- a static expression
```

ومثلما سنرى في الفصل ٨، يمكننا إعادة تعريف بعض (تحميل زائد) رموز العوامل، التي يمكن تطبيقها على أنواع معطياتٍ مجردة.

التعليمات (Statements) :

لا تملك ADA مجموعةً غنيةً من البنى لوصف المعطيات فقط، لكنها أيضاً، تحتوي على سلسلةٍ قويةٍ من التعليمات، لخلق خوارزميات. وبما أنّ ADA لغة برمجةٍ بنيوية، فإنها تقدم تعليماتٍ تسلسلية، تكرارية، وتحكماتٍ شرطية (بالإضافة لتعليماتٍ خاصة، مثل التعليماتِ المتعلقة بالمهام والإستثناءات). وتتضمن تعليمات ADA:

- تحكم تسلسلي: Sequential control:
  - الإسناد (Assignment).
  - الكتلة (Block).
  - لا شيء (Null).
  - العودة (Return).
  - استدعاء إجرائية (Procedure Call).
- تحكم تكراري: Iterative control:
  - الحلقات (Loops).
  - الخروج (Exit).

- التحكم الشرطي: Conditional control:
  - تعليمة Case.
  - تعليمة If.
- تعليمات مختلفة:
  - Abort.
  - ccept.
  - Code.
  - Delay.
  - Entry Call.
  - Goto.
  - Raise.
  - Select.

وستتم دراسة كل تعليمة بالتفصيل، بأحد الفصول٨، ٩، ١٤، ١٥. وتشكل تعليمة الإسناد، وتعليمة الإستدعاء، إجرائية بُنى التحكم التسلسلي الأساسية. وتعطي ***تعليمة الإسناد***، قيمةً محسوبةً جديدةً لمتغير:

```
Counter := Counter + 1;  -- a simple assignment
Birth_Day.Year := 1955; -- a record component assignment
```

وتسمح لنا البرامج الجزئية، بتنفيذ خوارزميةٍ مسمّاة. ومثلما سندرس في الفصل ٨، يمكننا ربط معاملات حقيقية وصورية، باستخدام ترميزات موضعية أو مسمّاة. فعلى سبيل المثال:

```
Start_Filling_Tank;             -- a procedure call
Value := Math_Functions.Tan(Angle); -- a function call
Rotate(Pints,30.0);     -- a procedure  call using positional notation
Rotate(Points, Angle => 17.6); -- a procedure call using
                -- named parameter associations
```

وهنالك العديد من التعليمات التسلسلية، ولكن سنؤجل مناقشة هذه التعليمات حتى الفصل ٩.

وتستخدم ADA تعليمتي التحكم الشرطي if, case. فتختار تعليمة case تعبيراً مقطعاً من بين عدة تعابير للتنفيذ، بينما تختار تعليمة if لا شيء، أو تعبيراً منطقياً من بين عدة تعابير للتنفيذ. وعلى سبيل المثال:

```
if Buffer_Size = Maximum_Buffer_Size then -- an if statement
 Process_Overflow;
end if;
if Value_Status = Open then  -- an if-then-else structure
 Read_Flow(Rate);
 else  Read_Pressure(Value);
end if;
case Buffer_Size is  -- selection from multiple paths
 when Maximum_Buffer_Size / 2 => Send_Overflow_Warning;
Get_New_Value;
 when Maximum_Buffer_Size      => Process_Overflow;
 when                          => Get_New_Value;
end case;
case Pixel_Color is  -- selection from multiple paths
 when  Red |  green | Blue  => Increase_Saturation;
 when Cyan..white          => Make_Black;
 when others               => null;
end case;
```

وإن التكرار مطروح في ADA، كواحدٍ من عدة أشكال تعليمة الحلقة Loop. وتسمح ADA بحلقة Loop أساسية، وحلقة (Counting)، وحلقة while. ويمكننا تطبيق تعليمة exit داخل حلقة Loop، لترك عملية التكرار. وعلى سبيل المثال:

```
loop               -- a basic loop
 Read_Modem(Symbol);
 exit when End_Of_Transmission;
 Display(Symbol);
end loop;
```

```
for Item in List'range  -- a counting loop
 loop
   Sum := Sum + List(Item);
end loop;

while Data_Available     -- a while loop
  loop
    Read(My_File, Input_Buffer);
    Process(Input_Buffer);
end loop;
```

وسنؤجل مناقشة تطبيق بقية تعليمات ADA بشكلٍ أساسيٍ على المهام ومعالجة الإستثناءات، إلى ما بعد.

البرامج الجزئية (Subprograms) :

مثلما رأينا حتى الآن، تحتوي ADA على عددٍ من الأنواع الأساسية، بالإضافة لآليةٍ، تسمح بخلق أنواع معطياتٍ مجردة (مستخدمين الأنواع الخاصة). وبلغة الخوارزميات المعينة، توازي البرامج الجزئية بـ ADA هذه المفاهيم، وذلك بتقديم آليةٍ لخلق عملياتٍ مجردة. وتعتبر البرامج الجزئية وحدة التنفيذ الأساسية في نظم ADA، ويمكن أن تكون واحداً من صفين:

- الإجرائيات (Procedures).
- التوابع الفرعية (Functions).

وتشبه البرامج الجزئية بقية الوحدات البرمجية بـ ADA (مثل الحزم البرمجية، والمهام)، إذ أنها تتألف من جزئين، توصيف، وجسم. ويُعرِّفُ التوصيف إسم البرنامج الجزئي، والمعاملات الصورية، ويعيد أنواع (من أجل التوابع الفرعية)؛ وهذا يمثل واجهة تخاطب الزبون للوحدة. وعلى العكس، يعلب جسم الإجرائية سلسلة التعليمات، التي تعرف الخوارزمية نفسها. وعلى سبيل المثال:

```
procedure Rotate(Points:inout Coordinate;Angle:in Radians) is
begin
 -- sequence of statements
end Rotate;
function Hash (Key : in Elements) return Hash_Value is
begin
 -- sequence of statements
end Hash;
function "*" (Left, Right : Matrix) returMatrix is
begin
 -- sequence of statements
end "*";
```

وتعين الحدود in, out ، in out نموذج، أو جهة، تدفق المعطيات، بالنسبة للبرنامج الجزئي؛ ويمكن للتوابع الفرعية، أن تتضمن النموذج in فقط. ولاحظ أيضاً، في المثال الأخير، كيف أعدنا تعريف رمز العامل "*"، لتطبيقه على المصفوفات، نوع معطيات مجرد. وسنفحص بنية تطبيقات البرامج الجزئية بـ ADA بالتفصيل في الفصل ٨.

الحزم البرمجية (Packages) :

تمثل الحزم البرمجية، إحدى الوحدات البرمجية الأساسية في ADA. وتسمح الحزم البرمجية للمستخدم، بتعليب مجموعة كيانات مرتبطة منطقياً (مجموعة موارد حسابية). وتدعم الحزم البرمجية في حد ذاتها مباشرةً، مبادئ البرمجيات، من تجريد معطيات، وإخفاء معلومات. وكما سنرى من خلال بقية الفصول (خصوصاً الفصل ١١)، هنالك طرق عديدة لاستخدام الحزم البرمجية، كما ينبغي. وكما في البرامج الجزئية تماماً، تتألف الحزم البرمجية، من توصيفٍ وجسم. ويشكل التوصيف، اتفاق المبرمج مع حزمة الزبون. ولا يحتاج الزبون مطلقاً لرؤية جسم الحزمة البرمجية، لأنه بالنسبة لجسم الحزمة، يتم إخفاء زرع التوصيف. فعلى سبيل المثال، لا يحتاج الزبون لمعرفة كيفية عمل التوابع الفرعية داخل حزمة برمجية، مع التوصيفات التالية:

```
package Graphics is
 type Turtle is private;
 procedure Set_Origin (A_Turtle     : in out Turtle;
                   X_Coordinate : in Integer;
                   Y_Coordinate : in Integer);
 procedure Turn (A_Turtle : in out Turtle);
 procedure Move (A_Turtle : in out Turtle);
 function X_Location (A_Turtle : in Turtle) return Integer;
 function Y_Location (A_Turtle : in Turtle) return Integer;
private
type Turtle is ...  -- completed type declaration
end Graphics;
```

ويأخذ جسم هذه الحزمة البرمجية، الشكل التالي:

```
package body Graphics is
procedure Set_Origin (A_Turtle     : in out Turtle;
                  X_Coordinate : in Integer;
                  Y_Coordinate : in Integer) is
begin
  -- sequence of statements
end Set_Origin;
procedure Turn (A_Turtle : in out Turtle) is
begin
  -- sequence of statements
end Turn;
procedure Move (A_Turtle : in out Turtle) is
begin
-- sequence of statements
end Move;
function X_Location (A_Turtle : in Turtle) return Integer is
```

```
begin
-- sequence of statements including a return
end X_Location;
function Y_Location (A_Turtle : in Turtle) return Integer is
begin
-- sequence of statements including a return
end Y_Location;
end Graphics;
```

وبهذه الطريقة، يمكننا فصل التوصيفات عن زرعها للتجريد.

المهام (Tasks) :

وتشكل المهام صفاً آخر من وحدات برامج ADA. وحتى هذه اللحظة، عرضنا البنى ذات الطبيعة التسلسلية. ومع ذلك، ففي النظم الحقيقية، يمكن لعددٍ من النشاطات المختلفة، أن تحدث بشكلٍ متوازي. فعلى سبيل المثال، في حالة تحكم إجراء، يمكن لنظامٍ أن يُظهر عشراتٍ من الحساسات المختلفة، ويقدم تقريراً عنهما مباشرةً إذا حدث شرط خروجٍ عن الحدود. ويجبر التكامل التسلسلي ترتيباً مؤقتاً لقراءة الحساسات؛ ويوجد عندها مجازفة، في ملاحظة بعض الحالات العابرة. وسيرى المنفذ معظم هذه الإجراءات، كمهام عديدة متوازية؛ وتسمح ADA بتجريدٍ مباشرٍ كهذا، ضمن اللغة.

وقد تمّ تأسيس نموذج مهمة في ADA، على مفهوم اتصال الإجراءات التسلسلي. وبمعنى آخر، يمكننا رؤية المهام كعملياتٍ مستقلة ومتوازية، يمكن أن تتصل مع بعضها البعض، بتمرير رسائل. وعندما تمرر مهمتان رسائل لبعضهما البعض، يقال عنهما أنهما في حالة *موعد* (*rendezvous*). والآلية الأساسية للإتصال، تتمثل من خلال تعليمتي مدخل (entry)، وقبول (accept). فعلى سبيل المثال، لنفرض أنّه لدينا مهمة تأخذ عينات من خط جهد، من ثم تمرر رسالة لمهمة أخرى، بعد أخذ عدة عينات. فالمهمة الداعية (Sampler)، تدعو المهمة الأخرى (Server):

```
task body Sampler is   -- body of the sampling task
begin
```

```
 loop
-- statements to get voltage samples
  Server.Report_On(Samples);
 end loop;
end Sampler;
```

ويجب أن تقبل المهمة المدعوة، دعوة المدخل:

```
task body Server is -- body of the receiving task
begin
 loop
-- sequence of statements
  accept Report_On(Samples : in Voltages) do
-- sequence of statements
  end Report_On;
-- sequence of statements
 end loop;
end Server;
```

وبشكل عام، فـإن الطريقـة التـي مـن أجلهـا تعـالج المهمـة المدعـوة العينـات، ستخفى بالنسبة للمهمة الداعيـة. وإن الأجـزاء الوحيـدة المرئيـة مـن المهمـة المدعـوة، ستوجد في قسم توصيف المهمة، مثل:

```
task Server is      -- the task specification
 entry Report_On (Samples : in Voltages);
end Server;
```

وفي أيّة حالة، إذا وصلت مهمة لنقطـة المدخـل، قبـل الأخـرى، فإنهـا سـتنام حتى تصل المهمة الأخرى، لنقطة الموعد.

وفي الحالة التي لا يرغـب فيهـا المصمـم لمهمـةٍ أن تنتظـر ، فإنـه يوجـد فـي ADA بُنى اختيارية، لذلك تتضمن:

- التأخير (Delay).
- المدخل المختار (Selective entry).
- الإنتظار المختار (Selective wait).
- المدخل المتزامن (Timed entry).

معالجة الإستثناءات (Exception Handling) :

في العديد من الحالات، تكمن النقطة الحرجة للأنظمة، في قدرتها على كشف شروط الخطأ، بسرعةٍ ولباقة..ففي معظم لغات البرمجة، يسبب مثلاً خطأً في توصيف الدخل، أو القسمة على صفر، توقف البرنامج، وتعيد التحكم لنظام الإستثمار. ويصبح هذا السلوك غير مقبول، من أجل النظم المحمولة، والتي تنفذ بشكلٍ مستقل عن تدخل الإنسان. وعلى العكس، فإن ADA تسمح بإمكانية معالجة الإستثناءات، في كتل بنيوية. والإستثناءات، التي تكون نموذجية (لكن ليست ضرورية) لشروط الأخطاء، يمكن أن تكون مسبقة التعريف (مثل الأخطاء الرقمية للقسمة على صفر)، أو معرّفةً من قبل المستخدم (مثل شرط طفحان الحافظة). وتبرز الإستثناءات المسبقة التعريف، بشكلٍ ضمني. فعلى سبيل المثال، إن تعليمات الإسناد التالية، ستبرز الإستثناء Constraint_Error، في زمن التنفيذ:

```
type Index is range 1..10;
Index_I : Index;
A_String : String (1..10);
Index_I := 0;             -- violation of type constraint
A_String(11) := 'x';      -- violation of index constraint
```

ويتم إبراز الإستثناءات المعرّفة من قبل المستخدم صراحةً، كما يلي:

```
procedure Pop (Element : out Integer; From : in out Stack) is
begin
 if Stack.Top = 0 then
  raise Stack_Empty;
 else
```

```
  -- statements
 end if;
end Pop;
```

فإذا حدث استثناء، سيتم ترك المعالجة الطبيعية، وينتقل التحكم لمعالج الإستثناء. وإذا لم يتواجد معالج الإستثناء، سيمر التحكم إلى مستوى المفردات (المُنفذ) التالي، إما إلى أن يتم الحصول على معالج استثناء، أو يتم الوصول إلى نظام الإستثمار (البيئة). وسندرس شكل وتطبيق تسهيلات استثناءات ADA في الفصل ١٥.

وحدات البرامج المولّدة (Generic Program Units) :

لقد أدرك مصممو لغة ADA، بأنّه يجب تطبيقها من أجل مشاريع برمجية ضخمة. وكنتيجةٍ لذلك، تم طلب ميزاتٍ تساعد على إدارة تعقيد هكذا نظم. ولقد ذكرنا منذ قليل، واحداً من هذه التسهيلات، التي تفصل وحدات الترجمة. وبالإضافة لذلك، تمنح ADA الوحدات المولّدة، كوسيلةٍ لبناء مركبات برمجية، يمكن إعادة استخدامها. فعلى سبيل المثال، في نظم معالجة مقاييس البعد، يمكن أن نحتاج لحافظة أغراض متعددة الأنواع، حيث ستكون هي خوارزمية معالجة هذه الحافظات نفسها؛ وفقط، أنواع المعطيات، هي المختلفة. ويمكن أن يأخذ المبرمج هذه الخوارزمية المشتركة، ويضع معاملاتٍ لها مع عناصر المعطيات، وبالتالي، خلق وحدة مولدة. ولاحظ بأنّ وضع المعاملات هذا، يُنجز في زمن الترجمة. فعلى سبيل المثال، إذا أخذنا الحزمة البرمجية Stacks، المطروحة سابقاً في هذا الفصل، وأضفنا جزءاً مولّداً، نحصل على:

```
generic  -- the generic part
  Limit : Natural;
  type Data is private;
package Stacks is
type Stack is private;
 procedure Push (Element : in Data; On    : in out Stack);
 procedure Pop  (Element : out Data; From : in out Stack);
private
 type List is array (1..Limit) of Data;
```

```
type Stack is
  record
    Structure : List;
    Top      : Integer range 0..Limit := 0;
  end record;
end Stacks;
```

ولا يخلق تعريف التوليد أي ترميز؛ فقط، يعرِّف قالباً للخوارزمية. ويجب على المستخدم أن يخلق نسخةً من وحدةٍ مولّدة، لخلق نسخةٍ محلية. وبالتالي، يمكننا خلق عدة أنواعٍ من المكادس:

```
package Integer_Stack is new Stacks (100, Integer);
package Float_Stack is new Stacks(Limit =>300,Data => Float);
```

ففي المثال الأول، يخلق التصريح حزمةً برمجيةً، منطقياً، مكافئةً، للحزمة البرمجية التي تمّ تعريفها سابقاً في هذا الفصل. ففي المثال الثاني، تمّ استخدام مجموعة المتغيرات المسماة، لتحسين قابلية القراءة. وسندرس المولدات بالتفصيل، في الفصل ١٢.

توصيف التمثيل (Representation Specification) :

وحتى عند البرمجة بلغةٍ عالية المستوى، من الضروري في بعض الأحيان، إستثمار بعض خصوصيات البنية الصلبة. وفي معظم لغات البرمجة، يريد أن يخلق المبرمج إجرائيةً منفصلةً، مكتوبةً بلغة المجمع، ومن ثم يربطها باللغة عالية المستوى. وتضمن ADA، على أيّة حال، وظائف تسمح بتحديد خصوصياتٍ تتعلق بالزرع، وتمثيل المعطيات. وبشكلٍ خاص، تسمح ADA بتوصيفاتٍ لما يلي:

- العناوين (Address).
- تمثيل النوع الرقمي (Enumeration type representation)
- الطول (Length).
- تمثيل نوع التسجيلة (Record type representation).

وعلى سبيل المثال:

```
for Printer_Status use at 16#177_776#;
    -- address specification
for Alarm  use (Normal => 0,    -- enumeration type
Temperature => 5,    -- representation
    Intrusion       => 57);
for Degress'Size  use 3*Bytes;  -- length specification
```

وسنفحص هذه التوصيفات، وخصائص برمجية منخفضة المستوى أُخرى، في الفصل ١٦.

الدخل/الخرج (Input/Output):

عادةً لا تتخاطب النظم المحمولة، مع أدوات الدخل/الخرج التقليدية، مثل الطابعة والطرفيات. وبدلاً من ذلك، يتم استخدام واجهات تخاطب خاصة "علب سوداء". ولمعالجة هكذا تنوع واسع من الأدوات، يتم الحصول على الدخل/الخرج بـ ADA من خلال عدة حزم برمجية. وبشكلٍ خاص، تحتوي ADA على حزمٍ برمجيةٍ مسبقة التعريف (لاحظ الملحق C) من أجل:

- الدخل/الخرج عالي المستوى.
- دخل/خرج غير نصي (دخل/خرج تسلسلي، ودخل/خرج مباشر).
- دخل/خرج نصي.
- الدخل/الخرج منخفض المستوى.

وبالتأكيد، يمكن للمبرمج أن يخلق الحزمه البرمجية الخاصة به، من أجل الدخل/الخرج. وكما أنّه لم ندرس بالتفصيل طبيعة الحزم البرمجية بـ ADA، فسنؤجل الفحص الكامل للدخل/الخرج، حتى الفصل ١٨.

٤ ـ ٤ ـ ملخص عن ميزات اللغة

(Summary of Language Characteristics):

تُظهر هذه اللمحة السريعة بأنّ ADA لغة مكتملة، ذات أهدافٍ عامة، وهي لغة عالية المستوى. وهي تؤكد على أهمية وثوقية البرنامج، وقابلية صيانته وقابلية تطبيقه في استخدام نظم برمجيةٍ ضخمة، ويمكن تغييرها بشكلٍ دوري. إن الميزة الأكثر أهمية في ADA، تتمثل بتجسيد مفاهيم الطرق البرمجية العصرية، وبالتالي، توفير أداةٍ فعليةٍ لمساعدة إدارة تعقيد الحلول البرمجية. وفي الفصول اللاحقة، سنفحص شكل واستخدام كل وسيلةٍ من ADA بالتفصيل. وقبل أن نترك هذا الفصل، دعنا ندرس بنية نظامٍ متكاملٍ في ADA، وبنفس الوقت، يجب تجميع الخطوات التي سنتبعها. فعلى سبيل المثال، لنعتبر تطبيقاً يجب تحقيقه على أعداد عقدية. فلا توفر ADA هكذا نوع مباشرة، ولكن من خلال استخدام الحزم البرمجية، مثل أنواع المعطيات المجردة، يمكننا تكوين تجريدٍ ملائم.

ويجب أن نسأل في البدء أنفسنا: ما هي العمليات، التي يمكن أن تخضع لها الأعداد العقدية؟. وبالتأكيد، يجب أن نقدم طريقةً لإعطاء، وإعادة إيجاد، قيم القسم الحقيقي، والقسم التخيلي، لقيمة عددٍ عقدي. وبالإضافة لذلك، سيكون من المفيد تقديم عمليات حسابية، مثل الجمع، والطرح. وبالتالي، يمكننا ترقيم العمليات كما يلي:

• Set	-- set the value of the complex number
• +	-- add two complex numbers
• -	-- subtract two complex numbers
• Real_Part	-- return the real part of the number
• Imaginary_Part	-- return the imaginary part of the number

وتشجع ADA استخدامنا للتجريد، بإعطائنا آلية لالتقاط قرارات تصميمنا، حول الأعداد العقدية. وبالتالي، يمكننا التعبير عن رؤيتنا للأعداد العقدية، بتوصيف الحزمة البرمجية:

```
package Complex is
```

```
type Number is private;
procedure Set (A_Number        : out Number;
            Real_Part       : in Float;
            Imaginary_Part    : in Float);
function "+" (Left, Right : in Number) return Number;
function "-" (Left, Right  : in Number) return Number;
function  Real_Part (A_Number : in Number) return Float;
function Imaginary_Part(A_Number : in Number) return Float;
private
 type Number is
  record
   Real_Part       : Float;
   Imaginary_Part : Float;
  end record;
end Complex;
```

ومثلما ناقشنا في بداية هذا الفصل، يوجد للحزم البرمجية قسمان. القسم المرئي من الحزمة (يمتد من السطر الأول، نزولاً حتى الكلمة المحجوزة private)، وهو يشير لكل شيءٍ مرئي بالنسبة لزبون الحزمة البرمجية؛ والقسم الخاص من الحزمة البرمجية (يتضمن الجسم)، ويشمل كل ما تبقى. وبهذه الطريقة، يظهر النوع Number إلى الخارج، كعددٍ عقدي، مع مجموعة عملياتٍ محدودة، ومعبرة. وعلى أية حال، فإن تمثيله الفعلي، مخفي عن جميع الزبائن، في القسم الخاص من الحزمة. وكما سندرس في فصلٍ لاحق، لا تسمح قواعد ADA لزبائن الحزمة البرمجية، أن يستخدموا أيّة معرفةٍ حول تمثيل النوع Number. وبهذه الطريقة، لا يمكن للزبائن انتهاك تجريدنا – الذي يبدو مفهوماً بسيطاً، ولكنه بالغ الأهمية، عندما نتعامل مع نظم ضخمة. وبعد كتابة هذا التوصيف، يمكننا إخضاعه للترجمة. وتذكّر، بما أنّه يمكن ترجمة الوحدات بـ ADA بشكلٍ منفصل، فمن الممكن أن نطوّر أجزاء النظام بشكلٍ متزايد، بدلاً من تطويره دفعةً واحدة. وبالتالي، يمكننا كتابة إجرائيةٍ تستخدم موارد هذه الحزمة

البرمجيـة. وبمـا أنّـه قـد تمّـت ترجمـة الحزمـة البرمجيـة Complex بشـكل منفصـل، يمكننـا أن نجعلهـا مرئيـة ببسـاطة، وذلـك، بتسـميتها فيمـا يُدعـى *توصيـف السـياق (context specification)*. وبالإضافة للحزمـة البرمجيـة Complex، سنسـتورد الحزمـة البرمجيـة المسـبقة التعريـف **Text_IO**، والتـي تُعطينـا تسـهيلات الدخـل/الخـرج. وبالتالي، يبدو الهيكل العظمي لهذه الإجرائية كما يلي:

```
with Complex, Text_IO;  -- context specification
procedure Calculate is
-- declarative part
begin
-- sequence of statements
end Calculate;
```

ودعنا نضيف جسم هذه الإجرائية. فلنفترض أننا نريد معالجـة عدديـن عقديـن، ومـن ثـم عـرض قيمـة أحدهـا. فبمـا أنّ ADA تتطلـب أن نعـرف كـل الأغـراض قبـل استخدامها، فإنه يجب أن نُضمن عدة تصريحاتٍ في جسم الإجرائية:

```
with Complex, Text_IO;procedure Calculate is
  First, Second : Complex.Number;
begin
  -- sequence of statements
end Calculate;
```

لاحظ كيف سمينا النوع First و Second . وهذا يعرف بإسمٍ مقيدٍ بشكل كامل، بما أنّه نريد أن نقيد الإسم من النوع، **Number**، مع إسم الحزمة البرمجية التي تصـدر **Complex**.

وبالتالي، يكون الإسم الكامل للنوع **Complex.Number**.

وبالتالي، نحتاج لبعض الطرق لكتابـة أعـداد مـن النـوع **Float**. وكمـا سـندرس مؤخراً بتفاصيل أكثر، فإن الحزمة البرمجية Text_IO، المسبقة التعريـف بــ ADA، لا تسمح مباشرةً بإجراء الدخل/الخرج، على الأنواع الممثلـة بالفاصلـة العائمـة، ولكنهـا توفر إمكانيات توليد.

وبالتالي، يمكننا إضافة النسخة المؤقتة:

```
with Complex, Text_IO;
procedure Calculate is
  First, Second : Complex.Number;
  package Complex_IO is new Text_IO.Float_IO(Float);
begin
-- sequence of statements
end Calculate;
```

ومن بين أشياء أُخرى، تُعطينا هذه النسخة طريقةً لطباعة أعدادٍ من Float باستخدام Put . ويمكننا إكمال جسم الإجرائية، بإضافة سلسلةِ التعليمات التي تُعطي قيمةً في البدء First و Second، ومن ثمّ تقييم تعبير يغير قيمة First، وأخيراً نعرض نتيجة First :

```
with Complex, Text_IO;
procedure Calculate is
 First, Second : Complex.Number;
 package Complex_IO is new Text_IO.Float_IO(Float);
begin
 Complex.Set(First, Real_Part => 5.7,Imaginary_Part =>-5.8);
 Complex.Set(Second,Real_Part=> 25.7,Imaginary_Part=> 18.35)
 First := Complex."-"(First, Second);
 Text_IO.Put('The real part of First is ');
 Complex_IO.Put(Complex.Real_Part(First),Aft => 2,Exp => 0);
 Text_IO.New_Line;
 Text_IO.Put('The Imaginarypart of First is ');
 Complex_IO.Put(Complex.Imaginary Part(First),Aft=>2,
Exp=> 0);
 Text_IO.New_Line;
end Calculate;
```

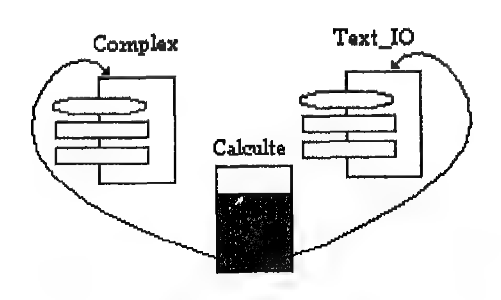
لاحظ بأنّ استدعاءنا للإجرائية Complex_IO.Put، تضمّن معاملاً من أجل عدد الأرقام بعد الفاصلة العشرية (Aft)، ونفس الشيء بالنسبة للقوة، ليتم عرضها (Exp) . أيضاً سيلاحظ القارئ الحريص بأنّ تجريدنا لـ First, Second يبدو غريباً. مثلما سنتعلم في فصل لاحق، فإن هذا هو نتيجة قواعد الرؤية في ADA. ويمكننا تبسيط هذا التعبير، بجعل عملية الطرح مرئيةً مباشرةً. فإذا أضفنا السطر التالي في قسم التصريح:

use Complex;

يمكننا عندها، كتابة تعليمة الإسناد كما يلي:

First := First - Second;

ويمكننا الآن، أخذ جسم هذه الإجرائية، وترجمته. ومثلما نرى في الشكل ٤ ـ ١٠، يتألف نظام ADA الحالي، من ثلاث وحدات مُترجمة بشكل منفصل. وعلى أية حال، فإننا لسنا جاهزين بعد لتنفيذ النظام. ومع ذلك، يمكننا بناءه مباشـرةً، على أساس تجريداتٍ موجودة (كما فعلنا بالنسبة للإجرائية Calculate، التي تتعلق بموارد Complex وText_IO)، ويجب أن نكمل جسم الحزمة البرمجية Complex قبل أن نُنفذ Calculate. ومثلما سندرس بتفصيلٍ أكثر لاحقاً، يجب أن يكمل جسم الحزمة البرمجية العناصر المقدمة في التوصيف. وبالتالي يجب أن يقدم جسم Complex، زرعاً لكل إجرائية، ولكل وظيفة، تمّ تحديدهما في قسم التوصيف. ويبدو الهيكل العظمي لـ Complex كما يلي:



الشكل٤ ـ ١٠. تبولوجيا نظام أعداد عقدية.

```
package body Complex is
 type Number is private;
 procedure Set (A_Number        : out Number;
            Real_Part       : in Float;
            Imaginary_Part : in Float) is ...
function "+" (Left, Right : in Number) return Number is ...
 function "-" (Left, Right  : in Number) return Number is ...
 function  Real_Part(A_Number: in Number) return Float is ...
 function Imaginary_Part(A_Number: in Number) return Float is ...
end Complex;
```

ويمكننا إكمال جسم كل برنامج جزئي، بقليلٍ من التعليمات البسيطة. وتذكر بأنّ جسم Complex، يكون منطقياً مخفياً من وجهة نظـر كـل زبـون، أي الإسـتقلالية عـن توصيف Complex، وعن الزبائن، وبالتالي، لدينا الحرية باختيار أيّ تنفيـذٍ يؤمـن لنـا السلوك الذي نأمله لتجريدنا. وبالحقيقة، من الممكن تغير جسم Complex، دون التأثير على التوصيف لأي زبون. ومن وجهة نظر عملية، فإن هذا يعنـي، أنّـه يمكننـا تطويـر نظامٍ ضخمٍ، بالبدء ببناء واجهات التخاطب لوحداتٍ مختلفة، ومن ثـم، تكـامل جميـع القطع، وفقط عندها، نبدأ بالبحث عن تنفيذ كل تجريد. وأكثر من ذلك، يمكننا محاولة عمليات تنفيذٍ جديدة، دون إعادة ترجمة الزبون لتجريدنا.

ولعرض هذا المثال بأكمله، دعنا نُضمن جسماً كاملاً، للحزمة البرمجية Complex:

```
package body Complex is
 procedure Set (A_Number      :  out Number;
            Real_Part      : in Float;
            Imaginary_part : in Float) is
 begin
  A_Number := (Real_Part, Imaginary_Part);
 end Set;
  function "+" (Left,Right : in Number) return Number is
  begin
```

```
  return (Left.Real_Part + Right.Real_Part,
        Left.imaginary_Part + Right.Imaginary_Part);
 end "+";
 function "-" (left,Right : in Number) return Number is
 begin
  return (Left.Real_Part - Right.Real_Part,
        Left.Imaginary_Part - Right.Imaginary_Part);
 end "-";

function Real_Part (A_Number : in Number) return Float is
 begin
  return A_Number.Real_Part;
 end Real_Part;

 function imaginary_Part(A_Number: in Number) return Float is
 begin
  return A_Number.Imaginary_Part;
 end Imaginary_Part;
end Complex;
```

ويمكننا الآن، ترجمة هذه الوحدة، لإكمال نظامنا. وبعد ربطنا وتحميلنا للنظام (كما هو مطلوب في تنفيذنا)، يمكننا تنفيذ النظام، بتنفيذ Calculate ، الذي يعمل وكأنّه الجذر، أو البرنامج الرئيسي، للنظام. ويؤدي تنفيذ هذا النظام، إلى الخرج التالي:

```
The real part of First is -20.00
The imaginary part of First is –24.15
```





5

مسألة التصميم الأولى:
الفهرسة الأبجدية للوثائق
Documents Concordance

تعريف المسألة

تحديد الأغراض

تحديد العمليات

تأسيس الرؤية

تأسيس واجهة التخاطب

زرع كل غرض



لقد تتبعنا تطور ADA، وأعطينا لمحة سريعة عـن إمكانياتهـا وشـكلها. وحتـى الآن، يجب ألا تتوقع أنك فهمت اللغة بشكل عميق، لكن يجب أن تكـون قـد كوّنـت فكرةً واضحةً عن بنيتها، وقوة تعبيرها. وفي الفصول الخمسة عشر التاليـة، سـنفحص ADA بالتفصيل، من وجهة نظر مبادئ هندسة البرمجيات، والطريقة غرضيـة التوجـه إذ تمتّ مناقشتهما في الفصلين ٢ و٣. وإن دراستنا لميزات لغـة ADA، سـتُكون مـن خلال حل خمس مسائل، تغطي مجالاً واسعاً من التطبيقات، متضمنة:

- الفهرسة الأبجدية للوثائق — الفصل ٥
- نظام قاعدة معطيات — الفصلان ٧ و١٠.
- حزمة برمجية لشجرة مولدة — الفصل ١٣.
- مراقبة البيئة — الفصل ١٧.
- إظهار الرأس مرتفعاً — الفصل ٢١.

وسنأخذ كل مسألة من خلال مراحل التصميم والتحليل، التي تقـود للتنفيـذ بـ ADA، وستحملنا هذه الطريقة على اختيار اللغة الصحيحة، من الأعلى للأدنى. وعندما نُكمل حـل كل مسألة، سنكتشف الحاجة لبعض التسهيلات داخل اللغة. وفـي تلـك النقطـة، سنعود خطوة إلى الوراء، وندرس هذه التسهيلات بالتفصيل. وسـيتجاوز الكتـاب قواعـد ودلالات اللغة. ففي الواقع، سنُركّز على الإستخدام الفعلي لـ ADA، ونقترح أسـلوب برمجـةٍ، يـبرز الوضوحية، وقابلية الفهم، ويشجع التجريد، وإخفاء المعلومات، والوحدوية، والمحلية.

٥ ـ ١ ـ تعريف المسألة (Define the Problem):

لنتذكر خطوات الطريقة غرضية التوجه، من الفصل ٣:

تحديد الأغراض.

تحديد العمليات.

تأسيس الرؤية.

تأسيس واجهة التخاطب.

زرع كل غرض.

ونحن الآن جاهزون لتطبيق هذه الطريقة، على مسألةٍ لتصميم نظامٍ يولّد فهرسةً أبجديةً لوثيقة. *والفهرسة الأبجدية، هي لائحة تضم جميع كلمات الوثيقة، مفهومة أبجدياً*، وتبين أماكن تواجد كل كلمة في الوثيقة. فعلى سبيل المثال، من أجل النص التالي:

we are ready to apply this method to the problem of designing a system that generates a concordance from a document. A *concordance* is an alphabetical index that shows the places in a document where each word may be found. For Example, A concordance for this paragraph might appear as:

سنحصل في الفهرسة الأبجدية، على ما يلي:

رقم السطر	الكلمة
1, 2, 3, 4	**A**
3	**Alphabetical**
3	**An**
1	**Apply**
4	**Might**

وتُستخدم الفهرسة الأبجدية بشكلٍ عام، كمساعد في دراسة الأعمال الضخمة، مثل الإنجيل، أو أعمال شكسبير الكاملة. وبشكلٍ مختلفٍ قليلاً، يمكن أن يستخدم نظام يستطيع خلق فهرسة لتوليد المراجع للبرامج، أو لخلق فهرس.

دعنا الآن، نعرض مسألتنا: نريد تطوير نظامٍ، يولّد فهرسةً أبجديةً كاملةً لوثيقة، بإعطاء إسم ملفٍ يحتوي هذه الوثيقة (الفهرسة الأبجدية الكاملة تجدول جميع الكلمات - بما فيها أدوات التعريف - وليس فقط الكلمات الأساسية). وسيُستخدم نظامنا في وثائق قصيرة، مثل البرامج؛ ومن أجل ذلك، نريده أن يتضمن رقم السطر الذي تظهر فيه كل كلمة، وليس رقم الصفحة، كما هو متبع في الكتب الضخمة. ومثلما سنرى، تبدو هذه المسألة معقدةً قليلاً، لكنها بسيطةً بشكلٍ كافٍ، لتكون أول مغامرةٍ لطيفةٍ في تطبيقات ADA.

٥ - ٢ - تحديد الأغراض (Identify the Objects):

كيف نبدأ بتقديم حلٍ برمجيٍ لمسألتنا؟ فطريقةٍ وظيفية، سنبدأ بتعين الخطوات الأساسية للإجراءات العامة: قراءة جميع الكلمات، ترتيب الكلمات، ومن ثمّ إنتاج تقرير. وعلى أي حال، كما تعلمنا في الفصول السابقة، تُصبح قيود هذه الطريقة بديهية، عندما نواجه مسائل أكثر فأكثر تعقيداً.

وبدلاً من ذلك، سنطبّق طريقة البرمجة غرضية التوجه، ونبدأ بتعين الأغراض الأساسية، التي تشكل نموذجنا لفضاء المسألة. فتذكر من الفصل الثالث، بأنّ الغرض، يمثل كياناً له حالة، ويمكن أن يُميز بالعمليات الخاضعة، والمحرِّضة. ومن مناقشتنا للمسألة، نحدد أربعة أغراض، أو صفوف أغراض:

الكلمات Words .

أرقام_الأسطر Line_Numbers .

الوثيقة Document .

الفهرسة الأبجدية Concordance .

فأي أسلوبٍ استخدمناه لتحديد هذه الأغراض؟ لاشيء، في الحقيقة. فإذا رجعنا لوصفنا للمسألة، نرى أنها تلك الأغراض، التي استخدمناها في وصف فضاء المسألة. وفي الحقيقة، يمكننا تعيين الأغراض في مجال المسألة، بطريقة غير صورية، ببساطة، بعزل الأسماء والجمل الاسمية التي استخدمناها لوصف المسألة. (مع ذلك، مثلما ناقشنا، فإن طرقاً أكثر صوريةً، تكون ضروريةً لنظمٍ معقدة).

وكل واحدٍ من هذه العناصر، يوافق معايير، قد بنيناها من أجل كل غرض. فأولاً، كل واحدٍ يمثل كياناً له حالة. وعلى سبيل المثال، توجد قيمة لكلمةٍ يمكننا رؤيتها، وتتألف الوثيقة من مجموعة كلمات، يمكننا استخراجها الواحدة بعد الأخرى. ومع ذلك، تجدر الإشارة إلى نقطةٍ هامة: يتحدث وصفنا للمسألة عن وثيقة واحدة، وفهرسة أبجدية واحدة، ولكن عن العديد من الكلمات والأسطر. وبما أننا نريد أن نعكس نموذجنا للحقيقة، بأكثر قدرٍ ممكن، سنأخذ بعين الاعتبار عدداً اختيارياً

للكلمات والأسطر. مثلما سنناقش في الفصل ١١، فإن هذا التمييز يفيد أكثر، في حفظ نموذجنا لفضاء المسألة، وكذلك، له تأثير على كيفية بنائنا للنظام. وبشكل خاص، سننمذج أغراضاً فردية، بما ندعوه *آلات حالات - مجردة (abstract_state machines)*: سنعالج أغراضاً عديدة، كنسخٍ من نوع معطياتٍ مجردة. وبمعنى آخر، تشير آلات حالات-مجردة، إلى أغراضٍ فردية، حيث تشير أنواع المعطيات المجردة، لصفوف أغراض.

فهل توجد أغراض أُخرى مهمة؟ *في هذا المستوى*، جوابنا لا. تذكر المناقشة في الفصل٢، وعلى أية حال: نرى العالم في مستويات التجريد. ويكون كل مستوى مفهوماً لوحده، لكنه يُبنى من تجريدات من مستويات أخفض. وبالتالي، ففي أعلى مستوى في نموذجنا للحقيقة، لا نرى إلا هذه الأغراض الأربعة فقط. ومثلما سنرى، تمّ بناء بعض هذه الأغراض، على تجريداتٍ بأخفض مستوى. وعلى أية حال، سنستخدم الآن مبدأ إخفاء المعلومات . بما أنّه لا توجد تفاصيل تنفيذية يجب أن تؤثر على نموذجنا الحالي لفضاء المسألة، سنؤجل هكذا تفاصيل، حتى تكون ضرورية لتنفيذ أخفض مستوى.

٥ ـ ٣ ـ تحديد العمليات (Identify the Operations):

لقد عينا الأغراض الأساسية الهامة، لكن هذا ليس كافياً لتثبيت بنية حلنا. وبالتالي، يجب علينا تعيين سلوك كل غرض. وبهذه الطريقة، نعطي معنى لكل تجريد؛ وبمعنى آخر، يجب أن نؤسس دلالة كل غرض.

وإن توجُّهنا بدلالة الأغراض، هو بشكل خاص، مفيد من أجل وجهة النظر هذه. وكل ما نريد عمله، هو تأسيس السلوك الخارجي لكل غرض. وتذكر بأنّه يمكن رؤية الأغراض بطريقتين: من الخارج، أو من الداخل. وتلتقط الرؤية الخارجية سلوك غرض من منظور زبائنه، بينما تعكس الرؤية الداخلية تنفيذ الغرض نفسه. وفي النقطة، التي نحن فيها من أجل معالجة حل المسألة، يجب أن نركز على الرؤية الخارجية. وطبقاً لمبدأ إخفاء المعلومات، سنؤجل تفاصيل الرؤية الداخلية لما بعد.

وبشكل عام، إن أفضل وسيلة لتأسيس مواصفات غرض، تتمثل بتعيين العمليات التي يخضع لها. وسنناقش هذه النقطة معمقاً في الفصل القادم، ولكن الآن، سندرس كل غرضٍ بدوره.

وأن الكلمة هي بدون شك، أعظم تجريدٍ رئيسيٍ في فضاء المسألة. ومن وجهة نظر الزبون، يمكننا تشكيل عمليتين على كلمة:

Create -- give a value to the word

alue_Of -- return the value of the word

قد تبدو هذه الخطوة بديهية، لكنها هامة جداً. ولاحظ بأننا لم ندقق على كيفية تمثيل الكلمة؛ ولكن بالعكس، على كيفية سلوكها المجرد. وبفصل سلوك الغرض عند تنفيذه بهذا الشكل، نكون قد طبقنا مبدأي التجريد، وإخفاء المعلومات.

وسنعمق هذه النقطة، لكن من الآن، لاحظ الإختلاف الطفيف بين هاتين العمليتين. العملية الأولى (Create)، تغير حالة كلمة. والعملية الثانية (Value_Of)، لا تغير الحالة، بل تعيد قيمة الحالة. وسيكون هذا الاختلاف مهماً لبنية حلنا، وبالتالي، سنصنف كل عملية، إما *بنّاءة* (Constructor) (عملية تغير حالة الآلة)، أو *مختارة* (Selector) (عملية تعيد قيمة لحالة غرض).

وإن رقم السطر، هو أيضاً تجريد بسيط. وبشكلٍ أساسي، يمكننا رؤية رقم سطر، كغرضٍ ذي قيمةٍ صحيحة. وبالرجوع إلى تعريف مسألتنا، نُدرك مع ذلك، بأنّه من الممكن إجراء تجريدٍ أكثر دقة (و مفضل). وبالحقيقة، لا معنى لأن نُعطي قيماً سالبةً، أو القيمة صفر، لرقم سطر. ومن الآن، سنجرد رقم السطر، كغرضٍ لا يمكن أن يأخذ إلا قيماً موجبة تماماً.

فما هي العمليات التي يمكن تطبيقها على رقم سطر؟ في هذه الحالة، نُبيح جميع التعابير الممكنة من أجل الأعداد الصحيحة؛ أي العمليات الحسابية العادية، من جمعٍ، وطرحٍ، ومقارنة. وإن ترقيم هذه العمليات ليس ضرورياً هنا، ولكنّ طبيعة تجريدنا للمسألة، يجب أن تكون واضحةً بشكلٍ كافٍ. مرة ثانية، لماذا كل هذه

الجهود؟ والجواب، بأنّه من المهم لنا بناء تجريدٍ مضبوط، يسمح لنا عند مواجهة المسائل الضخمة، بالحصول على التدريب الضروري لإدارة تعقيد مجالات مسائل أكثر صعوبةً.

ويتطلب التوثيق تجريداً أكثر صعوبة، حيث يمكننا وصف سلوك وثيقة، بالعمليات التالية:

Open -- open a document with the given name

Close -- close the document

Get -- get the next word and its line from the document

Is_End_Of_File --return True if there ano more words to read

لقد عرفنا ثلاث بناءات ومختار واحد. ومرة ثانية، بأية طريقةٍ تعرفنا على هذه العمليات؟ فبشكلٍ أساسي، فحصنا نموذج مسألتنا، واعتبرنا الأفعال التي يمكن أن تُطبق، وثيقة بالسّياق. وسيتعجب القارئ الحريص، لماذا جدولنا البناء «خذ» **Get** مع الوثيقة، بدلاً من أن يكون مع الكلمة (**Word**). والسبب هو أنّ «خذ» **Get** تُغير حالة الوثيقة، ولا تغير حالة الكلمة. ولذلك، ربطنا عملية ما مع الغرض التي تؤثر عليه، وليس مع الغرض الذي يكون، ببساطةٍ، غير فعال.

وبما أنّ الوثيقة تعتبر تجريداً أكثر تعقيداً، نتساءل عما يمكن أن يكون سيئاً في استخدامنا له. وهناك مبدأ أساسي سنطبقه، وهو أن جميع خواص الغرض، يجب أن تكون سليمة. وبمعنى آخر، نريد أن نخلق تجريداتٍ قوية. فعلى سبيل المثال، ماذا يحدث إذا حاولنا قراءة كلمة Get، من وثيقةٍ غير مفتوحةٍ من قبل؟ من الواضح، أننا نرغب بنوع جوابٍ معقول من الوثيقة نفسها. فمن المثالي أن يتم التأكيد، بأنه من المستحيل على زبون، أن يضع غرضاً في حالة غير منسجمة.

ومن أجل هذا السبب، كجزء من هذه الخطوة، سنعتبر أيضاً الشروط الاستثنائية، التي يمكن ربطها مع كل غرض. وبدلالة الوثيقة، يمكننا تعيين الشروط الممكنة التالية:

Open_Error	-- a document is already open
Close_Error	-- the document is already closed
Word_Too_Long	-- Get cannot process the next word
End_Of_File	-- Get is called when the file is already empty

ولكي يكون كاملاً، يجب أن نضيف، بأنّه لا توجد شروط استثنائية من أجل الكلمة (هذا تجريد بديهي). وفي حالة رقم السطر، يمكن أن يحاول الزبون إسناد قيمة، من خارج المجال الذي حققناه في صفات تجريدنا. ومن أجل أسبابٍ ستصبح واضحةً فيما بعد، سندعو هذه الشروط الاستثنائية بـ «تقييد ـ خطأ» Constraint_Error.

وتمثل الفهرسة الأبجدية ذاتها، آخر غرضٍ يبقى لنأخذه بعين الإعتبار. وقد جدنا بأنّه توجد ثلاث عمليات معبّرة فقط:

Start	-- initialize the concordance
Add	-- add a new word with its line number to the concordance
Make_Report	-- display the value of the concordance

ونشتق هذه الصفات من نموذجنا للحقيقة. فمن أجل تنفيذ الفهرسة الأبجدية، في البدء ننفذ Start للتقرير، ثم وبشكلٍ متكرر، نضيف الكلمات، وأرقام الأسطر الموافقة، إلى الفهرسة الأبجدية، و– عندما ننتهي– نستدعي Make_Report لعرض حالته. ولاحظ بأنّه لم نشغل أنفسنا بكيفية تنظيم الفهرسة الأبجدية. وبالتالي، وببساطة، نطلب بأنّ دلالة هذا الغرض، تتمثل بإضافة كلماتٍ جديدة، في أماكن ترتيبٍ موافق (وبالتالي، مع أرقام أسطرٍ موافقة). وعندما نُنتج تقريراً، نحصل على خرجٍ، بشكلٍ يلتقط المعلومات التي نحتاجها. ومرة ثانية، لقد طبقنا مبدأ فصل واجهة التخاطب والتنفيذ.

ومن أجل الفهرسة الأبجدية، نتوقع وجود شرطٍ استثنائي وحيد:

Overflow	-- no more words can be added

و لم نكن نقصد في عدم تضمين استثناء، يصف شرط إضافة كلمات، على فهرس أبجدي لم يتم إقلاعه بعد. لماذا؟ لأننا نريد أن نرى الفهرسة الأبجدية من

الخارج، ككيانات متسلسلة؛ وهذا يعني، أننا نرغب أن نكون قادرين على إضافة كلمات، وإجراء تقريرٍ في أية لحظة. وبالحقيقة، يمكننا إجراء تقريرٍ، قبل أن نضيف جميع الكلمات لوثيقة ما. وبواسطة الإقلاع الذي يتمثل تأثيره، ببساطة، في تنظيف الفهرسة الأبجدية، ووضعها في حالة مستقرة، يمكن لنا الحصول على حالة جديدة. ومثلما سنناقش كثيراً في الفصل ١١، فإن هذا النوع من الإقلاع، يكون مشتركاً في الحزم البرمجية، وكآلات حالات ـ مجردة.

٥ ـ ٤ ـ تأسيس قابلية الرؤية (Establish the Visibility) :

الآن، وقد وصفنا تصرفات كل غرض وصف أغراض، يجب أن نأخذ بعين الإعتبار كيف تتصل هذه الأغراض مع بعضها البعض. ومثلما ناقشنا في الفصل ٢، إحدى المشاكل المرتبطة بطرق التحليل الوظيفي، التي تتمثل بتوجيه تجريد المعطيات إجبارياً، ليكون عاماً؛ وهذا يعني، أن نجعل المعطيات مرئية، بالنسبة لجميع المستويات تحت النظام. وبالطريقة غرضية التوجه، نسعى بالضبط لعمل العكس. وبشكل خاص، نحاول الحد من رؤية الأغراض أكثر ما يمكن. ولا يُنتج هذا فقط وحداتٍ مرتبطة برخاوة (الذي يمثل سمةً مرغوبةً في النظم المعقدة)، ولكن يمكننا أن نحس بشكل أفضل، بعدم وجود الإرتباطات المرضيّة في نظامنا.

وبالتالي، فإن السؤال البسيط الذي يجب أن نرد عليه في خطوتنا هذه، هو، من أجل غرض محدد X، ما هي الأغراض الذي يعتمد عليها، وما هي الأغراض التي تعتمد عليه؟ فالإعتمادات التي تحدثنا عنها، هي الرؤية البسيطة بين التجريدات. وعلى سبيل المثال، يتطلب تجريدنا لوثيقة، أن تتألف من كلماتٍ وأسطر. وبالتالي، نقول، بأنّ الوثيقة تعتمد على صفوف الأغراض للكلمات، وأرقام الأسطر. وعلى أية حال - وهذه نقطة هامة جداً - فالعكس ليس صحيحاً. والكلمات وأرقام الأسطر، لا تعتمد على الوثيقة. وكنتيجةٍ لذلك، نستنتج بأنّ العلاقات بين الأغراض وصفوف الأغراض، تكون وحيدة الاتجاه. وبالحقيقة، مثلما سندرس مؤخراً، لا تشجع بنية ADA الدورانية بين الوحدات.

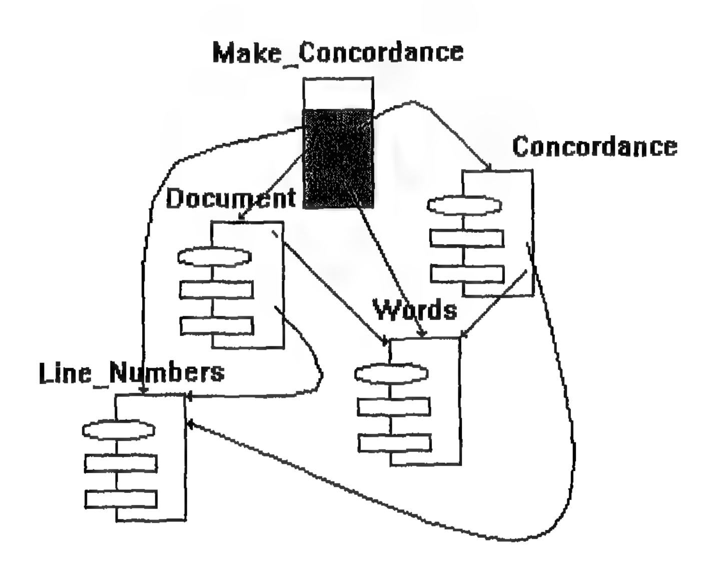
وفي هذا المستوى من التجريد، نلاحظ وجود الإعتمادات التالية:

تعتمد الوثيقة على الكلمات وأرقام الأسطر

(Document depends on Words and Line_Numbers)

تعتمد الفهرسة الأبجدية على الكلمات وأرقام الأسطر

(Concordance depends on Words and Line_Numbers)



الشكل ٥ ـ ١. بنية Make_Concordance ـ صنع ـ فهرسة أبجدية.

ومع ذلك، فإننا نحتاج لقطعة أخرى، من أجل ربط جميع أغراضنا مع بعضها البعض. وتتطلب ADA أن يكون جذر كل نظام، متمثلاً ببرنامجٍ جزئي. وبما أنّ هذا الجذر يمثل تجريداً خوارزمياً، سنسميه بجملةٍ فعليةٍ نشطة، **Make_Concordance**. ولا يمثل هذا الجذر بالضبط، غرضاً؛ وبالأحرى، فانه يُستخدم كوسيلةٍ تنسق نشاطات بقية الأغراض جميعها، في هذا المستوى. ومن أجل أسبابٍ ستصبح واضحة في المقطع التالي، توجد العلاقة الباقية:

تعتمد Make_Concordance على الكلمات، أرقام الأسطر، الوثيقة والفهرسة الأبجدية

(Make_Concordance depends on Words, Line_Numbers, Document, and Concordance).

ولقد قدمنا في الفصل ٤، العديد من الرموز، لتمثل وحدات ADA. فإذا طابقنا الأغراض Words, Line_Numbers, Documents, Concordance للحزم البرمجية، والوحدة Make_Concordance مع برنامج جزئي، فإنه يمكننا تمثيل بنية حلنا بالشكل ٥ ـ ١. ونرى هنا رمز جسم الإجرائية Make_Concordance، كبرنامج رئيسي. وإن جميع الوحدات الباقية، قد تمّ عرضها كتوصيف وحدات. ويتمثل سبب ذلك، بأنّه في هذا المستوى، فقط واجهات التخاطب لـ Words, Line_Number, Document, Concordance تكون مناسبة. وفي مقطعٍ متأخر، سنرى وجود رؤى أخرى، في مستويات تجريدٍ منخفضة. وهكذا، يمثل هذا الشكل بنية حلنا، في مستوى تجريد واحد فقط. وبالانتقال إلى زرع كل وحدة، يمكن أن نكتشف أغراضاً أو صفوف أغراض، نحتاجها من أجل زرعٍ واحدٍ من هذه الأغراض عالية المستوى.

ويمكن أن نستخدم الشكل ١ لتحليل تأثير التغيير على حلّنا. فعلى سبيل المثال، إذا قررنا تغيير التقرير الحالي المُنتج بواسطة الغرض Concordance، فيُظهر الشكل عدم تأثير أي وحدة أُخرى - ولا تعتمد أي وحدة على جسم الوحدة Concordance. ومن جهة أُخرى، إذا غيرنا واجهة التخاطب للحزمة البرمجية Line_Numbers، بعدم السماح لأن تكون أرقام الأسطر ذات قيمٍ سالبة، فإنّ كلاً من Document «الوثيقة» و Concordance و Make_Concordance ستتأثران، وبسبب اعتمادهما من وجهة نظر خارجية على Line_Number.

نرى بأنّ هذه الوحدات مرتبطة برخاوة، بسبب كون العلاقات بين الوحدات ذات اتجاه وحيد. ويمكننا رؤية Words و Line_Numbers بأنهما مجردتان تماماً وبشكل مستقل؛ ونفس الشيء يُطبق على Document و Concordance. ويُعتبر هذا

الفصل أساسياً، كلما انتقلنا إلى نظمٍ أكثر تعقيداً، لكي تُعطينا أعلى مستوى من الثقـة. إذ أن تغيراً في جزءٍ من النظام، لن يؤثر على أي جزءٍ آخر.

٥ ـ ٥ ـ تأسيس واجهة التخاطب (Establish the Interface):

بما أنّه قد خلقنا الآن تصميماً بأعلى مستوى لحلّنا، فإنـه يمكننـا اتخـاذ جميـع قرارات التصميم التـي حققناهـا حتـى الآن، علـى شـكل وحـدات ترجمـة بــ ADA. وبشكلٍ عام، فإننا نبحث عن ارتباطٍ صحيح، واحد–إلى–واحد، بين الأغراض (صفوف الأغراض) ووحدات ADA، لإنتاج وحدات مرتبطة برخاوة، وذات تماسكٍ عالٍ. وبهذه الطريقة، نُطبِّق ADA كلغـة تصميـم، لاتخـاذ العديـد مـن قـرارات تصميمنـا، حـول تصرف الوحدات في نظامنا. ومثلما تعلمنا في الفصول السابقة، نسـتطيع ترجمـة هـذه الواجهات بشكلٍ مستقل، لبناء نظامنا بشكل متزايد. وسنعتبر كل وحدةٍ فـي ترتيبهـا، الذي يجعلها صالحة للترجمة. (تذكّر، بأنـه يجـب أن يُـترجم توصيـف الوحـدة قبـل الرجوع إليه).

وإن **Line_Numbers**، الذي يشير إلى صـف أغـراض، يمكـن تمثيلـه بواسـطة توصيف حزمة برمجية بسيطة في ADA :

```
Package Line_Numbers is
   Type Number is range 1.. Integer'Last;
end Line_Numbers;
```

وتُصدر هذه الحزمة البرمجية نوعاً وحيداً، **Number**، حيـث تنتمـي قيمـه للمجال الذي يبدأ بـ ١، وينتهي بقيمـة صحيحـة ضخمـة (أكـبر قيمـة صحيحـة، يمكــن تمثيلهــا بشــكل وحيــد الدقــة). وبالتــالي، فــإن الحزمــة البرمجيــة **Line_Numbers**، تمثل صف أغراض. ومثلمـا سـنرى فـي المقطـع التـالي، فإنـه يمكننـا التصريـح عـن أغـراضٍ مـن النـوع **Number**، وضمنيـاً مـع هـذا النـوع مـن التصريح، يكـون التصريـح عـن جميـع العمليـات الحسـابية العاديـة، والتـي هـي بالفعل، الغاية التي نبحث عنها لأغراضنا.

ولاحظ كيفية تعليب الحزمة البرمجية نصياً، للتصريح عن Number. وأيضاً، إن هذه الخاصة، أكثر بداهةً في توصيف الحزمة البرمجية Words:

```
package Words is
  type Word is private;
procedure Create (The_Word      : out Word;
            With_The_Value : in String);
FunctiValue_Of (The_Word : in Word) return String;
Private
  type Word is ...
end Words;
```

ويوجد أيضاً هنا، نوع معطياتٍ مجرد. لكنَّ التصريح عن النوع Word، مختلف تماماً عن تصريح النوع Number. ومعنى النوع الخاص، يكون:

- مثيل النوع مخفياً من وجهة نظر خارجية.
- العمليات التي يمكن تطبيقها فقط على الأغراض من النوع هي تلك العمليات المجدولة في توصيف الحزمة البرمجية، مع الإسناد وفحص المساواة.

وبهذه الطريقة، نكون قد بنينا تجريداً جديداً، مطابقاً لاحتياجات فضاء مسألتنا. وتشجع قواعد لغة ADA تجريدنا، بإعطائنا آلية تسمح باتخاذ قرارات تصميمنا؛ وتقوي تجريدنا بإعاقة انتهاكه له. وبالحقيقة، بسبب كون النوع Word خاص، يُسمح للزبون فقط، تطبيق العمليات المرئية صراحةً، مثل الإسناد وفحص المساواة.

ومن أجل أسباب عملية زرع، فإن التمثيل الحقيقي لنوعٍ خاص، يجب إتمامه في القسم الخاص. وسنؤجل إتمام هذا الجزء، حتى نتعلم أكثر حول طبيعة أنواع ADA في الفصل التالي. (من أجل ترجمة هذه الوحدة، من الضروري إتمام القسم الخاص).

لاحظ كيف صرّحنا عن واجهة تخاطب كل عملية صريحة. وتسمح لنا قواعد لغة ADA، بكتابة فقط، توصيفات هذه البرامج الجزئية هنا، في توصيف الحزمة البرمجية. وقد تمّ تأجيل زرع أجسام البرامج الجزئية، إلى جسم الحزمة البرمجية. وفي توصيف الحزمة البرمجية هذا، لدينا إجرائية واحدة بمعاملين، تعمل كبنّاء، وتُقدم هذه العملية كلمةً، لها قيمة السلسلة المحرفية المعطية. والتابع الفرعي، الذي يُرجِع قيمة من النوع String، يعمل كمختار. والكلمتان المحجوزتان in ,out تدعيان بنماذج المعاملات – وهما تشيران إلى جهة تدفق المعطيات. فعلى سبيل المثال، إن المعامل The_Word في الإجرائية Create يحتوي النموذج out. وهذا يعني، أنّ الإجرائية لا تستخدم القيمة البدائية للمعامل؛ وبالتالي، فإن قيمة مُنتجة، تتدفق خارجةً من الإجرائية. وتتطلب قواعد ADA، أن تكون دائماً معاملات التوابع الفرعية من النوع in. وأخيراً، لاحظ بأنّه يجب علينا تعيين نوع كل معامل صراحة. وفي حالة المختار، فإن كل معاملٍ له نوع مختلف – واحد من النوع Word، والآخر من النوع String.

لنعتبر واجهة تخاطب Document. وتمثل Document غرضاً وحيداً، وهكذا، سننفذه كآلة حالة-مجردة. وبشكل خاص، هذا يعني أنّ الحزمة البرمجية التي تُشير إلى Document لا تصدر نوعاً؛ إنها تُصدر فقط، العمليات على الغرض. وبهذه الطريقة، فإن الحزمة نفسها تصلح كغرض – يمكننا ضمانة أنّ حلنا له بالضبط غرض واحد من هذا النوع. وبما أنّ الحزمة البرمجية Document تعتمد على تجريدات Words و Line_Numbers، يجب أن نُصرح عن هذه العلاقة بشكل صريح. وتوفر ADA وسيلةً لتحقيق ذلك، وهي *عبارة السياق*. ومثلما رأينا في الفصل السابق، يمكننا الإشارة إلى اعتمادٍ بين الوحدات، بتسمية الوحدة المعتمد عليها كجزء من عبارة with في بداية الوحدة. وبالتالي، يمكننا كتابة واجهة تخاطب Document كمايلي:

```
with Words, Line_Numbers;
package Document is
  Procedure Open (The_Name : in String);
```

```
  Procedure Cose;
  Procedure Get (The_Word    : out Words.Word;
            The_Number : out Line_Numbers.Number);
 Function Is_End_Of_File return Boolean;
 Open_Error        : exception;
  Close_Error       : exception;
 Word_Too_Long  : exception;
 End_Of_File       : exception;
 End Document;
```

لاحظ كيف يجب أن نُسمي النوعين Word, Number، وتكون القواعد من أجل الحزم البرمجية، مثل تلك التي يجب أن تعالج توصيفات الحزم البرمجية، مثل «الجلود skins» حول تجميع تصريحات. ونحصل على الوصول لحزمة برمجية تمّت ترجمتها بشكل منفصل من خلال عبارة سياق، لكن هذا لا يحقق الرؤية الآلية للتصريحات، داخل واجهة التخاطب لحزمة برمجية. أيضاً، إن الاسم الكامل من النوع Word يكون في الواقع Words.word؛ ويُطبق نفس الشيء على النوع Line_Numbers. Number .

لاحظ أيضاً، كيف عبّرنا عن الشروط الاستثنائية التي عرفناها من أجل الحزمة البرمجية Document – ببساطة نسمي هذه الشروط كاستثناءات. وبالتالي، على سبيل المثال، إذا حاولنا إغلاق وثيقة لم يتم فتحها، سيُصبح الاستثناء Close_Error نشطاً. وفي فصل لاحق، سنناقش طريقة اكتشاف هكذا شروط، وكيفية الرد عليها برمجياً.

وأخيراً، لنعتبر معنى النوع Boolean . وهذا نوع مسبق التعريف مع القيمتين True, False. إنّه مناسب تماماً لعمليات مثل Is_End_Of_File، فمن أجلها نريد معرفة شرط ثنائي.

آخر واجهة تخاطب لغرض يجب أن نعتبره يتمثل بالفهرسة الأبجدية Concordance. يمثل أيضاً آلة حالة. بما أنّ Concordance يعتمد على الوحدات

Words, Line_Numbers، يجب أيضاً أن نوفر عبارة سياق مناسبة With Words, :

Line_Numbers;__Package Concordance is__ procedure Start;__
procedure Add (The_Word : in Words.Word;__ The_Number :
in Line_Numbers.Number);__ procedure Make_Report;__ Overflow :
exception;
End Concordance;

بما أنّه لا يتم تصدير نوع خاص لا من Document ولا منConcordance ، فليس مناسباً لنا تضمين جزء خاص في توصيف الحزمة البرمجية وبالتالي، حتى تمثيل كل غرض يكون مخفياً عن الزبائن.

تُكمل واجهة التخاطب هذه طريقة اتخاذ قرارات تصميمنا من أجل كل وحدة إذا ترجمنا كل وحدة واجهة تخاطب بشكل عابر، نكون متأكدين بأنّ تجريداتنا ذات تماسك ملائم في هذا المستوى من التجريد.

٥ - ٦ - زرع كل غرض (Implement Each Object):

كخطوة نهائية، يجب تنفيذ كل غرض برمجياً .في الواقع، إذا كنا مهتمين فقط بتصميم نظامنا، يمكننا التوقف هنا .من أجل أن نكون كاملين سنعمق مناقشة الموضوع أكثر من ذلك بقليل لاحقا."

لقد أشرنا سابقا"، بأنّ ADA تتطلب أن يكون جذر كل نظام برنامجاً جزئياً، بالضبط هذا ما تمّ في الإجرائية . Make_Concordance لم ندرس بعد جميع تفاصيل تعليماتADA ، لكن دعنا نفحص جسم هذه الوحدة.

في جذر نظامنا، يجب أن تنسق Make_Concordancc نشاطات جميع الأغراض في أعلى مستوى تجريد .لقد ذكرنا منذ قليل الخوارزمية التي يجب أن نستخدمها هنا: إعطاء إسم الوثيقة وبعد ذلك بدء التقرير، نستخرج كلمات مفردة وأرقام الأسطر الموافقة من الوثيقة وحشرهما في الفهرسة الأبجدية . Concordance عندما تتم قراءة جميع كلمات الوثيقة، عندها نُنتج تقريراً.

الخوارزمية بسيطة ، لكن ADA تتطلب مادة أكثر لصقاً لتجميع كافة أجزاء الإجرائية مع بعضها البعض .مثلما فعلنا بـDocument وConcordance، يجب أن نوفر عبارة سياق لاستيراد جميع الوحدات التي يعتمد عليها مباشرة .أكثر من ذلك، يجب أن نوفر بعض التصريحات للأغراض من الأنواع Words.Word وLine_Numbers.Number. بما أنّDocument و Concordanceتمثلان آلات حالة ولا تصدران نوعاً، نحتاج عدم تضمين أي تصريحات أُخرى .أخيراً، نحتاج أن نُضيف تصريحين وتعليمتين لطلب إسم الوثيقة من المستخدم .يتطلب هذا النشاط أن نجعل الحزمة البرمجية المسبقة التعريف مرئية وذلك بتسميتها بعبارة سياق أُخرى. وبالتالي يمكننا التعبير عن جسم Make_Concordance كما يلي:

```
with Text_IO, Words, Line_Numbers, Document, Concordance;
procedure Make_Concordance is
  The_Name  : String(1..80);
  Last_Character : Natural; -- Values 0 .. Integer'Last
  The_Word  : Words.Word;
  The_Number  : Line_Number.Number;
Begin
 Text_IO.Put("Enter a document name: ");
 Text_IO.Get_Line(The_Name,Last_Character);
 Document.Open(The_Name(1..Last_Character));
 Concordance.Start;
 Loop
   exit when Document.Is_End_Of_File;
   Document.Get(The_Word, The_Number);
   Concordance.Add(The_Word, The_Number);
 end loop;
 Concordance.Make_Report;
 Document.Close;
end Make_Concordance;
```

وبسبب اصطلاحات التسمية التي استخدمناها، تبدو خوارزميتنا مقروءةً بشكلٍ تام، وبالضبط، توازي نموذجنا لفضاء مسألتنا.

وإذا أردنا أن نجعل إجرائيتنا أكثر قوة، يمكننا إضافة بعض الترميز، لمعالجة الإستثناءات الممكنة. فعلى سبيل المثال، في حالة نشاط الإستثناء Open_Error، يمكن عرض رسالة خطأً للمستخدم. ولتحقيق ذلك، سنُضيف ما ندعوه معالج الإستثناء. وسنستخدم أيضاً، الوحدة المسبقة التعريف Text_IO، التي توفر بعض التسهيلات، التي تجعل الخرج ذا قراءة مقبولة.

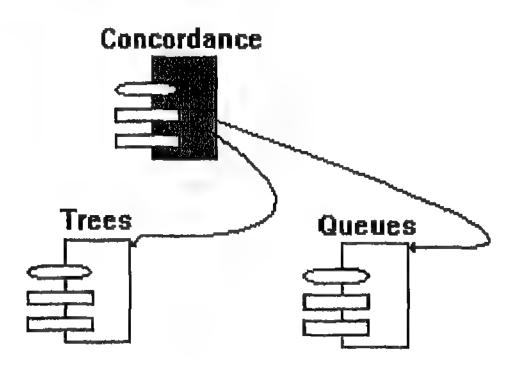
و بالتالي، فإن الهيكل العظمي لإجرائيتنا، يبدو الآن كما يلي:

```
with Text_IO, Words, Line_Numbers, Document, Concordance;
Procedure Make_Concordance is
  The_Name : String(1..80);
   ...
Begin
  Text_IO.Put("Enter a document name: ");
   ...
   Document.Close;
Exception
   When Document.Open_Error => Text_IO.Put_Line("Unable to
    open  the file.");
end Make_Concordance;
```

ويمكننا تطبيق طريقة مشابهة، من أجل بقية الشروط الاستثنائية.

إن نظامنا ليس جاهزاً للتنفيذ، لأنّه يجب علينا إتمام زرع كل حزمةٍ برمجية. ولحسن الحظ، فإن قواعد ADA هي تلك التي تمكننا من بناء وترجمة أجسام تلك الوحدات، بشكلٍ مستقل عن توصيفاتها. فبالعمل بهذه الطريقة، يمكننا محاولة زرع مختلف الأجسام، دون إعادة ترجمة أي واجهة تخاطب (التوصيفات). وهذه وسيلة هامة، كلما انتقلنا إلى نظمٍ أكثر تعقيداً.

ففي هذه الخطوة، تُصبح طريقتنا غرضية التوجه تراجعية. فكلّما نفذنا برمجياً كل غرض من مستوى تجريد عال، يمكننا مرة ثانية، تطبيق طريقةٍ غرضية التوجه، لاعتبار الأغراض التي تتركب منها الطريقة. وعلى سبيل المثال، كلّما بدأنا بتنفيذ الحزمة البرمجية Concordance، فإنه يمكننا اختيار تخزين كلمات شخصية مرتبة في شجرة ثنائية. وفي كل عقدة من الشجرة، نرتب الكلمة نفسها كمجموعة أسطر (أكثر دقة، رتل من الأسطر. وبالتالي، بالقفز عدة خطوات، نكون قد خلقنا مستوى التجريد المُبين في الشكل ٥ ـ ٢. وهنا نرى جسم Concordance، ولكن في هذا المستوى، يعتمد على غرضين جديدين هما Trees و Queues.



الشكل ٥ ـ ٢. بنية الفهرسة الأبجدية.

ولن نكمل التنفيذ البرمجي لكل جسم هنا؛ حيث يجب أن نحصل على مهاراتٍ إضافية في استخدام أنواع ADA وتعليماتها أولاً. وخلال الفصول القليلة التالية، سنهتم أكثر، في مشاكل التنفيذ البرمجي. وفي الفصل ١٧، نقدم أجسام هذا البرنامج.



6

تجريد المعطيات، وأنواع ADA

تجريد المعطيات

الأنـــــواع

التصريح بالأغراض



إن معظم اللغات البشرية، تبنى حول بنيتين أساسيتين: الأسماء والأفعال. أما العناصر الأخرى للغة، مثل الصفات، والظروف، فتستخدم لتضخيم أو إعاقة بنى الإسم والفعل. وجميع هذه العناصر، تشكل اللغة، التي تعتبر وسيلة الإتصال والتفكير. وسوف نتفحّص لغات البرمجة الأمرية، التي تُشعر الإنسان بنوع العمل مثلاً Push things on a stack,we Get our data، والحلقة loop لتكرار العملية. وكما لاحظنا في القسم الأول، فإن لغات الجيل الأول والثاني، تهتم بشكلٍ مبدئي بالتعابير والتحكُّمات، التي تخص بنى توجيه الأعمال، وفي الجيل الثالث، حيث بدء بفهم أهمية بنى المعطيات. وفي الواقع، فإن محاولة نقل أغراض العالم الحقيقي وعملياته، إلى عالم الحاسوب، يمكن أن تكون المساهمة الأكثر أهميةً، لأية لغة. وفي هذا القسم، سوف نفحص بشكلٍ كبير، فكرة التجريد، ونشاهد كيف تتفاعل لغة ADA مع تجريد المعطيات، وآلية التنويع Typing mechanisms.

٦ ـ ١ ـ تجريد المعطيات (Data Abstraction):

إن فكرة تجريد المعطيات، ليست جديدة. ففي الواقع، يمكن أن نقول، بأن كل ما نراه من أشياء في العالم الحقيقي هو مجرد. مثلاً، الكرسي الذي تجلس عليه، هو تجريد؛ وفي مستوى منخفض، يتشكل من ذرات. والكرسي، ماهو إلا عبارة عن إسمٍ نطلقه على مجموعة من الأشياء، لكي نستطيع التكلم عن خصائصها. وبشكلٍ مشابه، إذا كان لدينا بعض المعادن، والبلاستيك، والزجاج، مرتبةً بطريقةٍ خاصة، فإننا نعطي لهذه البنية إسم سيارة: والسيارة بصفتها تجريد، فإن خصائصها، تختلف عن مجموع خصائص كل من المعادن، والبلاستيك، والزجاج.

وإن تجريدنا للعالم، ليس بمستوى وحيد، بل إنه متعدد المستويات. وكلما صعدنا إلى المستوى الأعلى من التجريد، كلما شاهدنا منظراً أبعد، وكلما ابتعدنا أكثر عن التفاصيل المخبأة في المستوى الأدنى – والمثال التالي، يمكن أن يوضح هذه الطبقات من التجريد، حيث يمثل عدة مناظر مختلفة من القطة:

• Essence	(مستوى عالي جداً من التجريد)
• Living organism	(معظم تفاصيل المستويات الأدنى قد أهملت)
• Animal	(مع كل الكائنات المتحركة، غير المنتجة للغذاء)
• Cat	(صف واسع، حيث sam هو القضية)
• Sam	(إسم المجموعة الخاصة للأعضاء)
• Organs	(مجموعة ذرات، ذات مدلول بيولوجي)
• Atoms	(رؤية بدائية جداً للعالم)

وإن كل درجةٍ في هذا السلم، تمثل منظراً مختلفاً لنفس الغرض. ويمكن للسلم أن يتوسع في كل الإتجاهات.

فمن هذا المثال، يمكن أن نميز تجريدين. الأول يقول، بأنه لا يوجد مستوى معين لمعرفة رؤيتنا للعالم. حيث يرتبط المستوى الحالي من التجريد، مباشرةً بحاجتنا في هذه اللحظة. والطبيب البيطري، يهتم بالأعضاء (Organs)، بينما يهتم الفيلسوف بالغرض Essence.

وهذه الرؤية صحيحة، من أجل تطبيق معين. والثاني، هو أن كل مستوى يكون مزروعاً في مستوى أدنى. وهكذا، فإن تجريدٍ الـ Organ، يكون مزروعاً في مستوى أدنى بـ atoms، وتجريدنا لـ Cat ، يحدد في أدنى مستوى، بكيانٍ خاص إسمه Sam. والتجريدات، هي أكثر من أن تكون وسيلةً لوصف الأغراض الخاصةً. فهي تسمح في الواقع، بالتكلم عن خصائص صف أغراض. فكل مستوى، سوف يملك خصائص نوعية أكثر، لا يمكن أن تبنى ببساطة، اعتباراً من خصائص لمستوى أدنى. مثلاً، الحاسوب، عبارة عن تجريدٍ لمجموعةٍ من الذرات، ولكن هذا لا يعني أنه يمكن أن نتكلم، أو نصف عملية نظام الذاكرة الوهمية، في مستوى الجزيئات. وبشكلٍ مماثل ، لنأخذ مثالاً في استخدام معطياتٍ صحيحة. فالمعطيات الصحيحة، تزرع عادةً في مستوى منخفض في الحاسوب، مثل النظام الثنائي، الصفر أو الواحد. ولكن نادراً ما نفكر بهذا المستوى. ولذلك، نعتبر الخصائص الرقمية، لمثل هذه المجموعة من البتات (BITS). وهذا يسمح لنا بالتكلم عن الإضافة، والطرح، والجداء، والقسمة، وبقية العمليات المنطقية.

وأخيراً، إن التجريد يساعدنا في التركيز على المواصفات الهامة، وإهمال التفاصيل المملة. وكما هي الحال عند قيادة السيارة، فإننا لا نهتم بالتفاعلات الكيميائية الناتجة أثناء عملية الاحتراق الداخلي ضمن اسطوانة المحرك – التي تعتبر تفاصيل زرع الحركة.

وتستخدم عادةً، أدوات الكيان الصلب والبرمجيات، لبناء حلول مسائل العالم الحقيقي؛ وأما حلولنا هذه، فيجب أن تنمذج تجريدات العالم. مثلاً في وثيقة القسم الخامس، إن الأغراض الأساسية تكوِّن السطور، والكلمات، والوثائق. والتي يمكن أن نحركها، دون الحاجة إلى معرفة تمثيلها الداخلي، والشيء المهم بالنسبة لهذه الأغراض، هو خصائصها المنطقية. وفي مستوى معين، يمكن مع ذلك، أن تزرع هذه الأغراض كبنى معطياتٍ أولية، مثل المصفوفات، أو السجلات، أو المؤشرات. وفي أي مستوى ما، يمكن أن نعالج كل غرض، وكأنه غرض أولي.

وقد كانت تحتوي لغات الجيل الأول والثاني، بعض أدوات تجريد المعطيات. وكان يمكن بناء ذاكرةٍ مؤقتةٍ لمصفوفةٍ أو مؤشر، ولكن، كان يلزمنا وباستمرار، التحويل ذهنياً وباستمرار، بين التمثيل الفيزيائي من المستوى الأدنى لبنى المعطيات، وبين التمثيل المنطقي للذاكرة المؤقتة في المستوى الأعلى. وبشكلٍ عام، كانت بعض اللغات مثل لغة الباسكال، تعطي طرقاً أفضل لوصف المعطيات، ولكنها تبقى غير كافية لدعم الخصائص المنطقية. وفي الحالة المثالية في هذا الموضوع، من المحبذ للغتنا البرمجية:

- أن تجهز أدواتٍ لتوصيف بنى المعطيات المجردة.
- أن تدعم الخصائص المنطقية لتجريداتنا.

وفي القسم السابق، لاحظنا كيف يمكن لآلية الحزمة، المساعدة في تعريف التجريدات الجديدة، وكيف يمكن للحزم، أن تدعم الخصائص المنطقية. وبالرغم من ذلك، فإن كل تجريدٍ عالي المستوى، يجب أن يُزرع بواسطة عملياتٍ نوعيةٍ أولية، فمثلاً، تلك الأنواع الصحيحة أو الحروف. ومن خلال هذا القسم، سوف ندرس التسهيلات التي تقدمها «آدا» لتوصيف المعطيات الأولية.

٦ ـ ٢ ـ الأنواع (Types) :

في اللغات البشرية، تدعى الأشياءُ *أسماءً، وأما في لغة «آدا» نسميها أغراضاً.*
وكل غرضٍ له مجموعة من الخصائص (التي نسميها النوع، والنوع الجزئي)، التي تبين القيم التي يمكن أن يأخذها الغرض، من خلال العمليات المطبقة على ذلك الغرض. وفي نظام «آدا»، فإن الأغراض لا توجد بشكلٍ فطري (كما هو الحال بلغة الفورتران)، إلا إذا قمنا بالتصريح عنها بشكلٍ صريح، كما يلي:

```
Coefficient     : Float;
Count           : Integer;
Name            : String ( 1..80);
The_Tree        : Tree;
Water_Storage   : Tank;
```

ويجب التصريح عن أي كائن قبل استخدامه. وعندما نصرح عن غرض في «آدا»، فإننا نكتب إسمه أولاً، ومن ثم يتم ربطه بشكل الغرض. ولكن حتى الآن، يمكن أن نقول، أن التصريح عن غرض ما، يخلق نسخةً من النوع الخاص. مثلاً، إن النوع Integer، هو عبارة عن صفٍ من الأغراض، التي تبين مجموعة القيم والعمليات؛ و Count، هو عبارة عن إسم الغرض، الذي له نفس مواصفات النوع Integer.

ويقدم التنوع (Typing) آليةً لفرض بنى على الأغراض. والتنويع الصريح، هو أحد ميزات لغة «آدا»، وخاصةً في مبادئ البرمجيات الأساسية، المقدَّمة في الجزء الثاني. والتنويع يتطلب عدة حاجاتٍ مختلفة، أهمها:

- قابلية الصيانة *(Maintainability).*
- قابلية القراءة وسهولتها *(Readability).*
- الوثوقية (*Reliability*).
- تخفيض التعقيد *(Reduction of complexity).*

وبشكل عام، يتميز النوع بـ :

- مجموعةٍ من القيم.
- مجموعةٍ من العمليات المطبقة على الأغراض من النوع المعطى.

إن «آدا» لغة قوية النوع (*Strongly Typed*). وهذا يعني، أن الأغراض من نوع ما، يمكن أن تأخذ فقط القيم من هذا النوع، ويطبق عليها فقط العمليات المحددة لهذا النوع. وإنه لمن المهم ملاحظة، أن نوع الغرض هو Static، وهذا يعني، أن مواصفاته تحدد أثناء عملية الترجمة. بينما مواصفات النوع الجزئي، لا تعرّف حتى خلق الغرض.

وإن قوة النوع في «آدا»، تسمح لنا بالكشف عن كثيرٍ من الأخطاء خلال عملية الترجمة، وبالتالي، تسمح لمطور البرنامج، بتخفيض عدد الأخطاء أثناء التنفيذ.

وكذلك، إن ميزة قوة النوع في لغة «آدا»، تشابه التقدم المنطقي في سلّم التجريد. وحيث أن كل مستوى يبين قيمة مميزة، بالإضافة إلى مجموعة من العمليات المطابقة لذلك المستوى. مثلاً الاسم Apple، يؤشر إلى نوعٍ من أنواع الفواكه. فبالنسبة إلى الفلاح، فإن العمليات القابلة للتطبيق على هكذا نوع، هي: الزراعة، والجني، والتخمير، وعمليات أخرى. ولا تعني له أي شيء، أيٌ من عمليات الجمع والطرح (في المجال الرياضي).

وتتضمن صفوف أنواع ADA ما يلي:

- القيم السلمية، التي لا تملك مركبات.
- القيم المركبة، المؤلفة من أغراض مركبة.
- القيم المتصلة، التي تسمح بالوصول إلى أغراضٍ أخرى.
- القيم الخاصة، غير المعروفة من قبل المستخدم.

الأنواع السلمية (Scalar types):

تضم ما يلي:

- **الأنواع الصحيحة (Integer).**
- **الأنواع الحقيقية (Real).**
- **ا لأنواع المرقمة (Enumeration).**

وكما أشرنا سابقاً، تقسم أنواع المعطيات السلمية لعدة أنواع (الصحيحة، والحقيقية، المرقمة).

... أن توجد أنواع معرَّفة مسبقاً (a predefined type) أو ... (a user_defined type).

... الأنواع الصحيحة:

... من المعطيات، معرفٌ مسبقاً. وهو يشير إلى كافة الأعداد ... وهذا النوع، من الممكن أن يحتوي على أنواع ... مثل Long_Integer وShort_Integer (إن عدد الخانات الثنائية ... من الأنواع الثلاثة السابقة، مختلف عن عدد الخانات الثنائية، ...

... النوع من المعطيات، معرَّف مسبقاً. وهو يشير إلى كافة الأعداد الصحيحة ...

... النوع من المعطيات، معرَّف مسبقاً. وهو يشير إلى كافة الأعداد ... الموجبة تماماً.

... التعليمة التالية، يتم من قبل المستخدم تعيين Index، الذي يشير إلى ... المنتمية للمجال [١، ٥٠].

type Index is range 1..50;

...:

... نستعرض بعض الأنواع الحقيقية:

...: وهذا النوع من المعطيات، معرَّف مسبقاً. وهو يشير إلى كافة الأعداد ... الموجبة والسالبة.

type Mass is digits 10;

... التعليمة، تمّ تعريف النوع Mass من قبل المستخدم، الذي يشير إلى ... ممثلة بالفاصلة العائمة (floating point)، حيث العدد ١٠ يساعد في ... التمثيل.

العدد ١٠ يعرف عدد الخانات العشرية من الـ Significance.

يمكن تغير العدد ١٠ بأي ثابت صحيح موجب، وذلك حسب الحاجة.

type Voltage is delta 0.01 range -12.0..24.0;

فوفق هذه التعليمة، تمّ تعريف النوع Voltage من قبل المستخدم، الذي يشير إلى الأعداد الحقيقية، المنتمية للمجال الحقيقي [-12.0,24.0] ممثلةً بالفاصلة الثابتة. ووفق هذا التعريف، العدد 0.01، يمثل القيمة الفاصلة بين كل عددين حقيقيين متتاليين، الممكن تمثيلهما وفق هذا التعريف.

الأنواع المرقمة:

وفيما يلي، نستعرض بعض الأنواع المرقمة:

- **Boolean** : إن هذا النوع من المعطيات، معرَّف مسبقاً، وهو يشير إلى إحدى القيمتين المنطقيتين (False,True).

- **Character** : إن هذا النوع من المعطيات، معرَّف مسبقاً، وهو يشير إلى مجموعة المحارف المعرفة وفق ترميز الـ ASCII.

Type Color is (Black,Red,Green,Blue,Cyan);

فوفق هذه التعليمة، تمّ تعريف النوع **Color** من قبل المستخدم، والذي يضم مجموعة الألوان المحددة بين القوسين.

Type Card_Suit is (Clubs, Diamond, Hearts, Spades);

ووفق هذه التعليمة، تمّ تعريف النوع **Card_Suit** من قبل المستخدم، والذي يضم أنواع ورق اللعب.

ويمكن تعريف نوع مرقم عناصره مزيج من عدة أنواع، مثال ذلك ما يلي:

Type Mix is (Left,'L',Right,'R');

فوفق هذا، قد تمّ تعريف Mix كنوع مرقم عناصره الكلمة Left، والحرف L والكلمة Right والحرف R.

ويمكن تطبيق عدة عمليات على الأنواع المرقمة ، وهذه العمليات ، ملخصة بالجدول التالي:

Set Of Value	An Order Set Distinct Value	
Structure	(E0,E1,....,En)	Where Ei is An Ordered Enumeration Literal
Set Of Operations	Assignment	:=
	MemberShip	in Not in
	Qualification	
	Relational	= /= < <= > >=
Attributes	Address	Pred
	Base	Size
	First	Succ
	Image	Val
	Last	Value
	Pos	Width
Predefined Types	Boolean	
	Character	

وسيتم شرح هذه العمليات في المكان المناسب.

فمثلاً Card_Suit'First=Clubs و Card_Suit'Last=Spades

هذا، وإن الأنواع الصحيحة، والأنواع المرقمة، تدعى بالأنواع المتقطعة.

الأنواع المركبة (Composite types):

تقسم أنواع المعطيات المركبة إلى قسمين، وهما ما يلي:

- المصفوفة (**Array**).
- التسجيلة (**Record**).
- **المصفوفة (Array)** : فالمصفوفة، مؤلفة من مجموعة عناصر، لها نفس النوع.
- **التسجيلة (Record)** : والتسجيلة، مؤلفة من مجموعة عناصر، ليست بالضرورة من نفس النوع.

أمثلة:

أنواع المصفوفات :

وفيما يلي بعض الأمثلة، عن كيفية تعريف أنواع المصفوفات:

type chess_board is array(1..8,1..8) of color;

فوفق هذه التعليمة، قد تمّ تعريف النوع **chess_board** على أنّه مصفوفة مربعة، أبعادها 8*8، وعناصرها من النوع **color**، الذي تمّ تعريفه سابقاً، وأدلتها تنتمي لـ [1,8]*[1,8].

type pixel is array(Color) of float;

ووفق هذه التعليمة، قد تمّ تعريف النوع pixel على أنّه مصفوفة أحادية البعد، عدد عناصرها ٥ و هي من النوع الحقيقي float، وأدلتها تنتمي للمجموعة {black,Red,Green,Blue, cyan}.

type sensor is array(Index range 5..10) of Voltage;

ووفق هذه التعليمة، قد تمّ تعريف النوع sensor على أنّه مصفوفة أحادية البعد، عناصرها ٦ وهي من النوع Voltage المعرّف سابقاً، وأدلّتها تنتمي للمجال [5,10].

type vector is array(positive range<>) of Integer;

ووفق هذه التعليمة، قد تمّ تعريف النوع vector على أنّه مصفوفة أحادية البعد، غير محددة الطول، عناصرها من النوع الصحيح، وأدلتها موجبة.

type simple_array is array(positive range <>,positive range <>) of float;

ووفق هذه التعليمة، قد تمّ تعريف النوع simple_array على أنّه مصفوفة ثنائية البعد، أبعادها غير محددة، وعناصرها من النوع الحقيقي، وأدلتها موجبة.

```
Type Extended_Index is range 0..1_000;
Type Long_Array is array(Extended_Index) of float;
Type Short_Array is array(Extended_Index range 10..49) of float;
Type Limited_Array is array(Extended_Index range <>)of float;
```

ووفق الأربع تعليمات هذه قد تمّ تعريف:

- النوع **Extended_Index**، كمجال من الأعداد الطبيعية، تنتمي قيمه للمجال [0,1000].

- النوع **Long_Array**، كمصفوفة من الأعداد الحقيقية، تنتمي أدلتها للمجال Extended_Index.

- النوع **Short_Array**، كمصفوفة من الأعداد الحقيقية، تنتمي أدلتها لجزء من المجال Extended_Index، وهي محدودة بالمجال [10,49] .

- النوع **Limited_Array**، كمصفوفة من الأعداد الحقيقية، تنتمي أدلتها لجزء من المجال Extended_Index، حيث يتم تحديده عند اللزوم. (لاحظ كيفية التصريح عن Array).

وفيما يلي، تلخيص لبنية المصفوفات، وللعمليات الممكن تطبيقها عليها:

Set Of Value	An Indexed Collection Of Similar Types.	
Structure	Array (Index{,Index}) Of Component (Unconstrained Array)	Where Index{,Index} is a series of unconstrained discret types; the component denotes the type of values the array can hold
	array index_constraint of component (Constrained array)	where index constraint is a list of discrete types; the component denotes the types of values the array can hold
Set Of Operations	Adding (One Dimensional Array)	&
	Aggregate	
	Assignment	:=
	Explicit Conversion	
	Indexing	
	Logical (Boolean Components)	And Or Xor Not
	Membership	In Not in
	Qualification	
	Relational	= /=
	Relational (Discrete Components)	< <= > >=
	Unary (Boolean Components)	not



ـ Open : من النوع boolean.

ـ Flow_Rate : من النوع الحقيقي، وينتمي للمجال [0.0,30.0].

ـ Inspected : من النوع Date المعرّف في المثال السابق.

```
Type Cpu_Flags is
   Record
      Carry       : Boolean;
      Interrupt   : Boolean;
      Negative    : Boolean;
      Zero        : Boolean;
   End Record;
```

ووفق هذه التسجيلة، قد تمّ تعريف النوع **Cpu_Flags**، وهو مؤلف من مؤشرات وحدة المعالجة المركزية في الحاسوب.

```
Type Cpu_State is
   Record
      Priority : Positive;
      Flag     : Cpu_Flags;
   End Record;
```

ووفق هذه التسجيلة، قد تمّ تعريف النوع **Cpu_State**، وهو مؤلف من:

ـ Priority من النوع Positive.

ـ Flag من النوع Cpu_Flags

وتمثل هذه التسجيلة، حالة وحدة المعالجة المركزية في الحاسوب.

```
type square(side:positive:=4) is
  record
    Matrix : simple_array(1..side,1..side);
  End record;
```

ووفق هذا، قد تمّ تعريف النوع square على أنّه مصفوفة مربعة، أبعادها تتحدد بالقيمة الموجبة side. وهذا النوع، من النوع simple_array المحدد بأمثلة المصفوفات، وإذا لم تعين قيمة side فإنّ side تأخذ القيمة ٤، المحددة ببداية تعريف التسجيلة.

```
type aircraft_id is (Civilian,Military,Foe,Unknown);
type Aircraft_Record(kind:Aircraft_id:=Unknown) is
      record
        Airspeed   : speed;
        Heading    : direction;
        Latitude   : coordinate;
        Longitude : coordinate;
        Case kind is
         When Civilian  => null;
         When Military  => Classification: Military_type;
                                      Source     :Country;
          When Foe | Unknown => Threat     : Threat_Level;
    end case;
  end record;
 Aircraft : Aircraft_Record
```

ووفق هذا المثال، قد تمّ فيالبدء تعريف النوع المرقم Aircraft_id، وبعد ذلك، تمّ تعريف النوع **Aircraft_Record** كتسجيلة تتحدد بعض مركباتها وفق الوسيط kind، الذي هو من النوع Aircraft_id ، وإن مكوّنات التسجيلة ما يلي:

- Airspeed : من النوع speed، الذي نفترض أنّه معرّف.
- Heading : من النوع direction، الذي نفترض أنّه معرّف.
- Latitude : من النوع coordinate، الذي نفترض أنّه معرّف.
- Longitude : من النوع coordinate، الذي نفترض أنّه معرّف.

مثال:

ليكن Aircraft غرض من Aircraft_Record، ونريد التصريح عنه من أجل kind=Military، مع إعطاء مركباته القيم البدائية التالية:

Airspeed=150.0
Heading=97.3
Latitude=147.6
Longitude=27.1
Classification=Transport
Source=France

فيتم التصريح عن ذلك، كما يلي :

```
Aircraft : Aircraft_Record(Military) :=(Airspeed  => 150.0,
                                        Heading        => 97.3,
                                        Latitude       => 147.6,
                                        Longitude      => 27.1,
                                        Classification => Transport,
                                        Source    => France );
```

أما بقية المكوّنات فتتحدد وفق قيمة kind، وهي التالية :

- إذا كانت قيمة kind تساوي Civilian، فلا يوجد مركبات إضافية.
- وإذا كانت قيمة kind تساوي Military، عندها يتم تعريف المركبتين :
- Classification : من النوع Military_type، الذي نفترض أنّه معرّف.
- Source : من النوع Country، الذي نفترض أنّه معرّف.
- وإذا كانت قيمة kind تساوي Foe أو Unknown عندها يتم تعريف المركبة :
- Threat : من النوع Threat_Level الذي نفترض أنّه معرّف.

وإذا لم يصرح عن قيمة kind، فإنها ستأخذ القيمة Unknown.

وفيما يلي، جدول يلخص بنية التسجيلة، والعمليات الممكن تطبيقها عليها :

Set Of Values	A Collectio Of (Potentially) Differently Named Componenets	
Structure	Record Component_List End Record;	Where Component_List Declares The Elements Of The Record
Set Of Operations	Aggregate	
	Assignment	:=
	Explicit Conversion	
	MemberShip	In Not In
	Qualification	
	Relational	= /=
	Selection	

Attributes	Record Types:
	Address
	Base
	Constrained
	Size
	Record Components:
	First_Bit
	Last_Bit
	Position

أنواع الوصول (Access types):

حتى الآن، استعرضنا الأنواع من أجل الأغراض الثابتة التي تعرف وقت الترجمة. وهناك عدة حالات، يجب أن تخلق معطيات الأغراض خلال وقت التنفيذ (dynamically). وفي ADA، قيم الوصول، تشير إلى أغراض أخرى، وتسمح لنا بخلق أغراض ديناميكياً.

مثال:

```
type buffer is
 record
  Message : string(1..10);
  Priority  : positive;
 end record;
type buffer_pointer is access buffer;
```

ففي هذا المثال، buffer_pointer، يمثل نوع الوصول، وهو يشير إلى الأغراض من النوع buffer.

ويمكن التصريح عن عدة أغراض متغيرة من هذا النوع، كمايلي:

```
pointer My_Packet, Your_Packet,Their_Packet : buffer_;
```

وتأخذ جميع هذه الأغراض في البدء، القيمة Null. ويمكننا خلق غرض جديد من النوع Buffer، باستخدام الكلمة المحفوظة New، كما يلي:

```
My_Packet    := New Buffer;
Your_Packet  := New Buffer'(Message=>'*********',Priority=>1);
Their_Packet := New Buffer'("---------',10);
```

ووفق هذه الأمثلة الثلاث، قد تمّ خلق ثلاث أغراض جديدة من النوع Buffer، إذ أنّه في المثال:

- **الأول**، تمّ خلق غرض جديد دون إعطائه قيمة.
- **الثاني**، تمّ خلق غرض جديد وإعطائه قيمة، وفق طريقة مجموعة المتغيرات المسماة، بحيث Message=********* وPriority=1.
- **الثالث**، تمّ خلق غرض جديد وإعطائه قيمة، وفق طريقة التمثيل الموضعي، بحيث Message=--------- وPriority=10.

ويمكننا خلق غرض من النوع Buffer، ومن ثم حذف كل ما يتعلق به، على الشكل التالي:

```
My_Packet := New Buffer;  --- a Buffer Object is Created
My_Packet := Null;   --- The Original Object is UnReachable
```

ويجب التمييز بين قيم الوصول، وأغراض الوصول. على سبيل المثال:

```
My_Packet:= New Buffer'(Message=> "+++++++++", Priority=> 1);
          --  Create A Buffer Object
Your_Packet := New Buffer'(Message=>"++++++++++",Priority=>10);
          -- Create Another Buffer Object
Their_Packet := Your_Packet; --  Point To The Same Object
```

و بالتالي، فإن العمليات التالية صحيحة على المؤشرات:

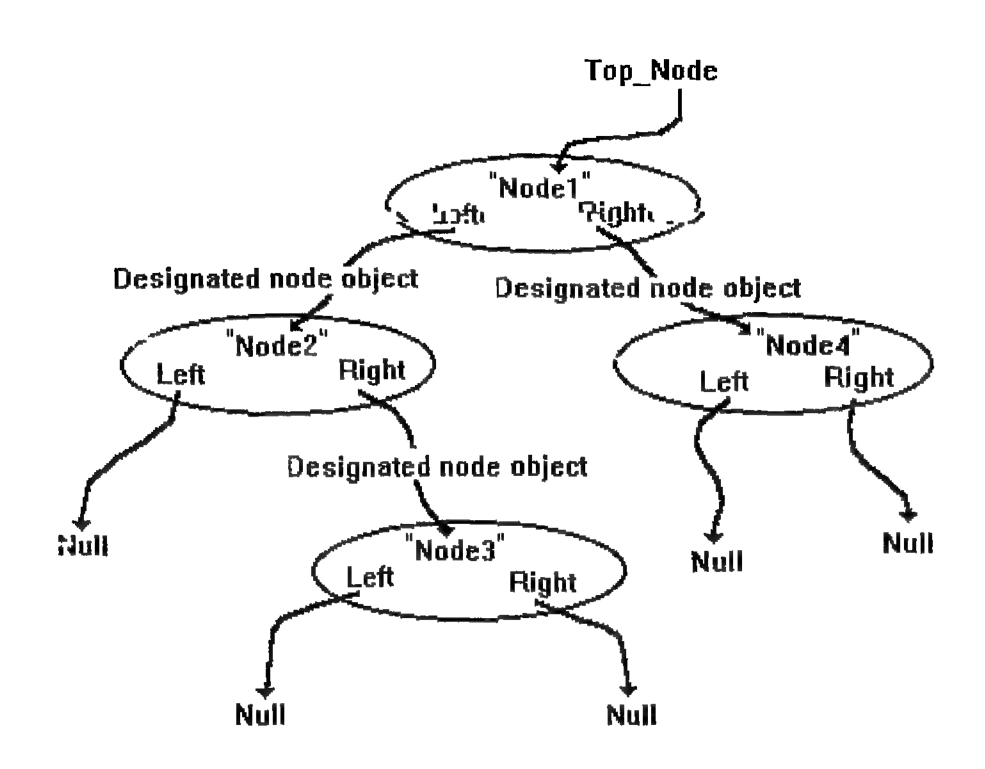
```
Your_Packet /= My_Packet  --- They Point To Different Objects
Your_Packet=Their_Packet  ---  They Point To The Same Object
Your_Packet.all /= My_Packet.all --  The Object Are Not Equal
Your_Packet.Message=My_Packet.Message
          --- Components Have Equal Values
```

ويمكننا استخدام قيم الوصول، لوصف العلاقة بين الأغراض، وخصوصاً، إذا تم تغيرهما مع مرور الزمن. وهذا مشابه تماماً لبنية معطيات لها شكل «سلسلة مترابطة» (Linked Lists) أو «شجرة ثنائية» (Binary Tree). فعلى سبيل المثال، يمكننا كتابة برنامج لشجرة ثنائية، كما يلي:

```
Type Node;
Type Tree Is Access Node;
Type Node is
 Record
  Left     : Tree;
  Value    : String(1..5);
  Right    : Tree;
 End Record;
Top_Node, Temp_Node, Pointer : Tree;
```

وباستخدام الوصول للأغراض، يكون البرنامج الذي يمثل العلاقات بين الأغراض، كما يلي :

```
Top_Node           := New Node;          -- Create The First Node
Top_Node.Value     := "Node1";           -- give it a value
Temp_Node          := New_Node;          -- Create Another Node
Temp_Node.Value    := "Node2";           -- give it a value
Top_Node.Left      := Temp_Node;         -- Link the nodes
Temp_Node          := New Node;          -- Create a third node
Temp_node.value    := "Node3";           -- give it a value
Pointer            := Top_node.left;     -- Point To Node2
Pointer.Right      := Temp_Node;         -- Link the nodes
Temp_Node          := New Node;          -- Create a fourth node
Temp_Node.Value    := "Node4";           -- give it a value
Top_Node.Right     := Temp_Node;         -- Link the nodes
```

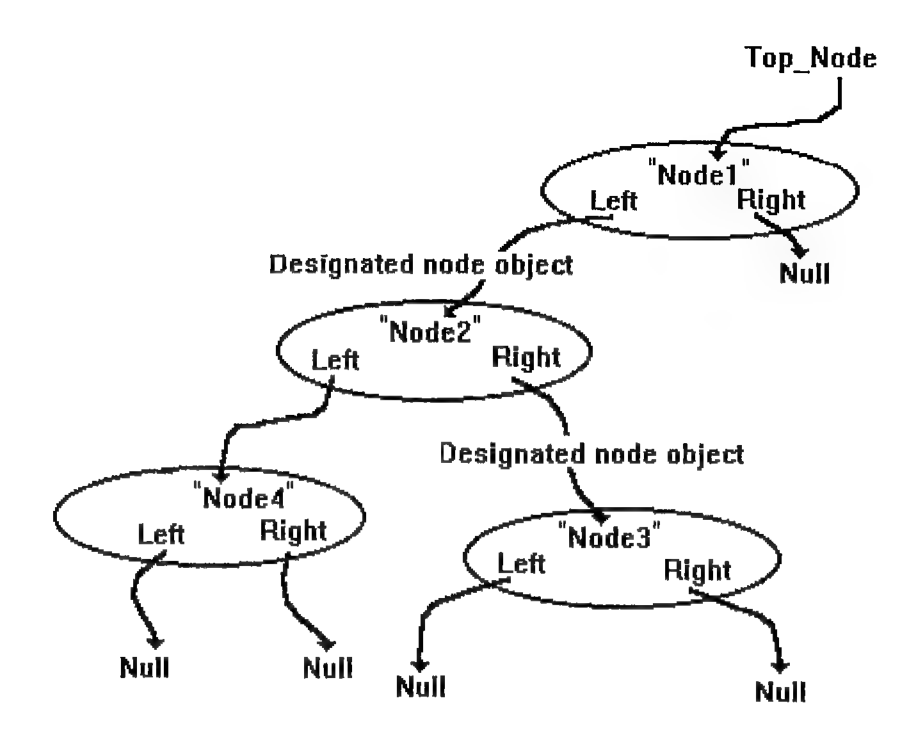


الشكل ٦ ـ ١. العلاقات بين الأغراض.

ويمكننا تغيير العلاقة بين هذه الأغراض، وفق التعليمات التالية :

```
Pointer := Top_Node.Left;            -- Point To Node2
Pointer.Left:= Top_Node.Right;       -- Access Nod4
Top_Node.Right := Null;              -- Break The Duplicate link
```

وتصبح الشجرة الثنائية بعد التغيير، وفق الشكل التالي:



الشكل ٦ ـ ٢. تغيير العلاقات بين الأغراض.

ويمكن الإستعاضة عن مجموعة التعليمات التي تمثل الشكل الأول، بالتعليمة التالية :

```
Top_Node := New Node'(Value => "Node1",
            Left => New Node'(Null,"Node2",
                New Node'(Null,"Node3",Null)),
            Right => New Node'(Null,"Node4",Null));
```

ملاحظة :وفق هذه التعليمة، وعند خلق العقد، قد تمّ استخدام طريقة التمثيل الموضعي وطريقة مجموعة المتغيرات المسمّاة.

والجدول التالي، يمثل بنية أنواع الوصول، وبعض العمليات الممكن تطبيقها عليها :

Set Of values	Access Values To designated Objects	
Structure	Access SubType_Indication	Where Subtype_Indication is The Type Of The designated object
Set Of Operations	Allocation	
	Assignment	:=
	Explicit Conversion	
	Indexing	
	(Array_Designated Objects)	
	Membership	In Not In
	Selection	
	(Record_Designated Objects)	
	Qualification	
	Relational	= /=
Attributes	Access Type:	
	Address	
	Base	
	Size	
	Storage_Size	
	Array_Designated Objects:	
	First	
	First(N)	Where N is The Nth Index
	Last	Range
	Last(N)	
	Length	
	Length(N)	Task_designated Objects:
	Range	Callable
	Range(N)	Terminated

الأنواع الخاصة (Private types):

تستخدم أنواع المعطيات هذه فقط، بالحزم البرمجية (packages) في قسم التوصيف حصراً، وهي تعرِّف مجموعة قيم، ومجموعة عمليات قابلة للتطبيق، وبنية أنواع المعطيات الخاصة، غير مرئية بالنسبة للآخرين، كما أنّه يمكن تعريف بعض العمليات على الأنواع الخاصة التي تصبح العمليات الوحيدة التي يمكن استخدامها. والأنواع الخاصة، تعتبر وسيلةً لإخفاء المعلومات، وخلق أنواع معطيات مجردة جديدة.

وفي الحزمة البرمجية، تأخذ الأنواع الخاصة الشكل: إسم النوع، متبوعاً بكلمة Private أو عبارة Limited Private. وبعد ذلك، يتم توصيف البرامج الجزئية، التي ستستخدم الأنواع الخاصة وغيرها، كأغراض دخل وخرج متغيرة.

مثال: فيما يلي يتم تعريف مكدس مجرد:

```
package stacks is
 type stack is private;
 procedure Push(Element : in Integer;On : in out stack);
 procedure Pop(Element : out Integer; From : in out stack);
 private
  Maximum_Elements : constant Integer :=100;
  type List is array (1..Maximum_Elements) of integer;
  type stack is
   record
    structure : List;
     top      : Integer range 0..Maximum_Elements :=0;
   end record;
 end stacks;
```

والجدول التالي، يمثل بنية الأنواع الخاصة، والعمليات الممكن تطبيقها عليها:

Set Of Values	Hidden From The User	
Structure	Hidden From The User	
Set Of Operations	Explicit Conversion	
	Membership	In Not In
	Qualification	
	for limited private, only those operations defined in the	
	Corresponding package specifications are true; for private	
	types ,assignment and tests for equality and inequality are	
	also available.	
Attributes	Private Types:	
	Address	
	Base	
	Size	
	Private Type With discriminants:	
	Constrained	

الأنواع المشتقة، والأنواع الجزئية (Subtypes and Derived Types):

الأنواع الجزئية (Subtypes) :

لا يعرِّف النوع الجزئي أي نمط جديد، بل يعرِّف إسماً جديداً، لنوع معطيات محدد سابقاً.

مثال١ : وفق التعريف التالي:

Type Month_Name is (January , February, March ,April , May , June , July August , September, October,November,December);
Subtype Summer is Monthe_Name range June..August;
Current_Month : Month_Name;
Vacation_Time : Summer;

لقد تمّ تعريف النوع الأساسي Month_Name، كنوع مرقم، وهو مؤلف من أسماء أشهر السنة بالترتيب. وبعد ذلك، تمّ تعريف النوع Summer، كنوع جزئي من النوع Month_Name

و الذي يضم أسماء الأشهر التالية: June, July, August. وبالتالي، لم نخلق نوعين منفصلين من أنواع المعطيات، إذ أنّه يمكننا أن ننفذ عمليات على أغراض من النوعين معاً.

مثلاً، العمليات التالية صحيحة:

Current_Month := Vacation_Time;

Vacation_Time := Summer'Succ(Current_Month);

ففي الحالة الثانية، إذا كان Current_Month لا يأخذ إحدى القيم May, June, July، فإنه سيكون هناك خطأ ناتج، وهو Constraint_Error، وذلك بسبب إسناد قيمة إلى Vacation_Time ليست من النوع الجزئي Summer.

ويمكن أن نفحص فيما إذا كان Current_Month، ينتمي إلى النوع الجزئي Summer أم لا، كمايلي: **If Current_Month in Summer then ..... End if;**

وإن عنصرا الفحص In, Not In معرَّفان من أجل جميع الأنواع، وليس فقط من أجل الأنواع السلّمية.

والأنواع الجزئية مفيدة، إذا كان تجريدنا للمسألة، يعرِّف عدة مجموعات من النوع الأساسي.

```
Type Non_Negative is range 0..Integer'Last;
  Location : String(1..20);
Type Weight is delta 0.01 range 2000.0..3000.0;
Type Vector is array(natural range <>) of float;
Type Sensor_Class is (Humidity, Pressure, Tempreture);
Type Sensor(Kind: Sensor_Class) is
 Record Case Kind is
    When Humidity     => Humid   : Natural;
    When Pressure     => Press   : Float;
    When Temperature => Temporary : Integer;
   End Case;
 End Record;
```

ومن هذه الأنواع، يمكن تعريف الأنواع الجزئية التالية:

```
Subtype Index is Non_Negative range 0..10;
   -- Range Constraint
Subtype Coarse_Weight is Weight delta 10.0;
   -- Accuracy Constraint
Subtype Vector_3d is Vector(1..3);
   -- Index Constraint
Subtype Heat_Sensor  is Sensor(Kind => Temperature);
   -- discriminant Constraint
```

ومن أجل كل حالة من الحالات، يمكن التصريح عن أغراض من النوع الأساسي، والنوع الجزئي. وبشكل حر، يمكن تنفيذ عملية على أيّ منهما، بشكل لا يكون هنالك خطأ ناتج بسبب حدود النوع الجزئي. فمثلاً يمكن إضافة غرض من Index، إلى غرض من Non_Negative، وإسناد الناتج إلى غرض من Index شرط ألا يكون الناتج أكبر من ١٠، وإلا سيكون هنالك خطأ وهو Constraint_Error.

ولاحظ أنّ تعريف أيّ نوع جزئي، يجب أن يتقيد بشروط النوع الأساسي، فمثلاً:

```
Subtype Illegal_Index is Non_Negative range -1..1;
```

إن هذا التعريف خطأ، لأن 1- لا ينتمي للنوع الأساسي Non_Negative.

وفي بعض الأحيان، يمكن تغيير إسم النوع الجزئي فقط، ونحصل بالتالي على نوع جزئي جديد، مثلاً:

Subtype Not_Negative is Non_Negative;

ويمكننا أيضاً، تعريف نوع جزئي من نوع جزئي آخر، كما يلي:

Subtype Big is Non_Negative range 0..1_000_000;
Subtype Small is Big range 0..10;

فوفق هذا المثال، قد تمّ تعريف Big، على أنّه نوع جزئي من النوع الجزئي Non_Negative، وبعد ذلك تمّ تعريف Small، على أنّه نوع جزئي من النوع الجزئي Big.

وللنوع، كملاحظة أخيرة، إذا أردنا معرفة أكبر قيمة في النوع Index، يمكن أن نستخدم الواصف Base، كما يلي: Index'Base'Last. وأيضاً الواصف Base، يمكن أن يستخدم لتحديد الدقة للنوع الأساسي، المختار من الأنواع الحقيقية، وعدة عوامل أخرى. وكمثال على ذلك، ما يلي:

Type Mass is Digits 15;
....
Masse'Base'Digits;
Mass'Basc'First;
Masse'Base'Last;

الأنواع المشتقة (Derived Types) :

بشكل غير مشابه للأنواع الجزئية، فإن الأنواع المشتقة، تعرّف عدة أنواع. وهذا ضروري عندما نجرد المسألة، ونلاحظ أنها تحتوي أغراضاً مختلفةً لها نفس البنية.

مثال: لنعرّف Mass وweight، على أنهما نوعين من النوع Float، كما يلي:

Type Mass is new Float;
Type Weight is new Float;

فوفق هذا التعريف، إنّ Mass وweight، نوعان مشتقان من النوع Float، الموجود من أصل اللغة، ولكن اللغة لا تسمح لنا، وبشكل «مضمر» (Implicitly)، بالمزج بين أغراض من النوعين Mass وweight. فمثلاً، لا يمكننا إضافة قيمة غرض من النوع Mass، إلى قيمة غرض من النوع Weight، بشكل مضمر.

وبشكل أساسي، إن النوع المشتق، يرث جميع صفات النوع الأب، بما في ذلك مجموعة العمليات، والواصفات، ومجموعة القيم، ورموزه (Literal)، وكل شيء.

والنوع المشتق، ينتمي لنفس الصف الذي ينتمي إليه النوع الأب. وأيضاً، إن النوع المشتق، يمكن أن يملك قيوداً غير قيود النوع الأب، بشكل مشابه للنوع الجزئي. ومثال، ذلك ما يلي :

Type Budget is new Float range 0.0..12_000.0;

فوفق هذا، قد تمّ تعريف Budget كنوع مشتق من النوع Float، بالإضافة لذلك، فإنه محدد بالمجال [0.0,12000.0] ، وإن لغة «آدا» ADA، في هذه الحالة، تجبر أغراض النوع Budget، لتأخذ قيماً حقيقيةً تنتمي للمجال المذكور.

فليكن لدينا التصريحان التاليان:

Software_Budget : Budget;
Total_Budget : Float;

فيمكننا وبشكل صريح، تحويل نوع الغرض Software، إلى نوع الغرض Total_Budget، كما يلي:

Total_Budget := Float(Software_Budget);

وبشكل عام، إن أي نوع يمكن أن يشتق منه، إذ يجب أن يكون نوع الأب قد اكتمل تعريفه، قبل الإشتقاق منه. وهذا يعني، أنّه لا يمكن الإشتقاق من نوع لم يكتمل تعريفه بعد. وأيضاً، الأنواع الخاصة، لا يمكن الإشتقاق منها، حتى بعد اكتمال تعريف النوع الخاص، في القسم الخاص من الحزمة البرمجية.

ومثلما نوهنا سابقاً، بأنّ النوع المشتق يرث جميع العمليات الممكن تطبيقها على النوع الأب.

فإذا كان النوع الأب يمثل:

- أحد الأنواع المعرفة مسبقاً، فالعمليات المورّثة، تمثل جميع العمليات المسبقة التعريف.
- أنواع تمّ تعريفها من قبل المستخدم، فالأغراض التي تنتمي للنوع المشتق، ترث جميع البرامج الجزئية المرئية بالنسبة لها، والخاصة بالنوع الأب، أو أحد أنواعه الجزئية.

وبالطبع، يمكن للمستخدم تعريف برامج جزئية، خاصة بالنوع المشتق. مثال ذلك ما يلي:

```
Type date is ...
Type My_Date is New date;
function "+"(Left : in date; Right : in date) return date is ...
procedure julian (A_date: in date; number: out integer) is ...
```

وبالنسبة للنوع المشتق My_date، نجد أنّ البرامج الجزئية "+" julian هي برامج جزئية مشتقة (أن أي غرض من النوع المشتق، يمكنه استخدام البرامج الجزئية المشتقة)، ولكن إذا صرحنا عن أي عمليات (برامج جزئية) أخرى، بعد التصريح عن My_date، فلن تكون هذه البرامج الجزئية مشتقة.

وإن إمكانيات الأنواع المشتقة، تبدو أفضل عند اشتقاق الأنواع الخاصة. والمثال على ذلك ما يلي: نريد اشتقاق بنية معطيات، لمكدّسٍ من بنية معطيات لقائمة مترابطة.

ولنفترض أنّ القائمة المترابطة، معرّفة على الشكل التالي:

```
Package Linked_Lists is
Type List is Private;
Function "&"(Left : in Integer; Right : in List) Return List;
Function "&"(Left : in List; Right : in Integer) Return List;
Function Head(Of_List : in List) Return Integer;
Function Tail(Of_List : in List) Return List;
```

```
Private
   Type List is ....
End Linked_Lists;
```

وبما أنّ المكدّس يمكن أن يُرى كقائمة مترابطة، بحيث يمكن تنفيذ جميع العمليات على نهاية من القائمة، يمكن اشتقاق المكدّس من القوائم المترابطة، كما يلي:

```
With Linked_Lists;
Pacakge Stacks is
  Type Stack is Private;
  Procedure Push (Item : in Integer; Onto : in out Stack);
  Procedure Pop  (Item : Out Integer ; Off_Of : in out Stack);
Private
   Type Stack is new Linked_List.List;
    -- At this Point all the operations of the type
    -- List are defined for the type stack.
End Stacks;
Package body Stacks is
  Procedure Push(Item : in Integer; Onto : in out Stack) is
    Begin
      Onto := Item & Onto;
    End Push;
  Procedure Pop (Item : out Integer; Off_Of : in out stack) is
    Begin
     Item := Head(Off_Of);
     Off_Of:=Tail(Off_Of);
    End Pop;
End  Stacks;
```

٦ ـ ٣ ـ تصريح عن الأغراض (Object Declarations) :

في ADA، يمكن إلتماس تجريد فضاء المسألة من خلال الأنواع، حيث أن الأنواع تشكل قالباً. ويجب أن نصرِّح عن الأغراض التي يستخدمها برنامجنا. وبالطبع، كنا قد صرَّحنا في الأمثلة السابقة عن الأغراض، ولكن يوجد بعض التفاصيل التي سنذكرها هنا.

وتدخل الأغراض في جزء التصريحات من البرنامج، أو البرامج الجزئية، ووحدات البرامج المولّدة. وعندما نصرّح عن غرض ما، يجب أن نحدد، وبشكل صريح، نوعه. مثلاً :

```
Distance  : Float ;
Response  : Character ;
Number    : Integer ;
Grades    : array ( 1..100) of Float ;
```

لاحظ في المثال السابق، أن النوع Grades لا يوجد له إسم، وإنما خلقنا نوعاً غير معروف.

كما يمكن إضافة قيود على الأغراض أثناء التصريح. مثلاً:

```
Name    : String (1..40);
Bottom  : Integer range -10 .. -1;
```

وينصح عادةً باستعمال الأنواع المشتقة، عوضاً عن استخدام القيود.

وكما ذكرنا سابقاً، تعتبر ADA لغةً قوية الأنواع، وهذا يعني، أن الأغراض من نوعية مختلفة لا يمكن أن تتفاعل بسهولة. ولذلك، تستخدم ما يسمى بالأنواع المتكافئة، حيث يكون الغرضان من نوع واحد، إذا أدخلا بنفس التسمية. مثلاً :

```
type Distance is digits 4 ;
type Length   is digits 4 ;
Width     : Length ;
Extent    : Distance ;
```

Width وExtent غرضان من أنواع مختلفة، ولكن هذه الأنواع لها نفس البنية .

ويمكن إعطاء قيمة أولية كقيمة بدائية، كما في المثال التالي :

```
Extent : Distance := 0.0 ;
```

إن لغة ADA، لا تعرف أية قيمة أولية وبدائية للأغراض (ما عدا قيمة Null، فإنها تعطى عادةً كقيمة بدائية للأغراض المتصلة) . وهذا يعود للمبرمج، لإعطاء القيم البدائية للأغراض قبل استخدامها. ويمكن إسناد هذه القيمة أثناء التصريح عن الغرض، كما فعلنا ذلك مع Extent.كما يمكن تطبيق طريقة ثانية على أنواع التسجيلة، وذلك

بتعريف قيم مركبات الغرض. فمثال ذلك، ما قمنا به في الفصل ٤ في مركبة Top من الغرض Stack، حيث تأخذ القيمة الأولية التي تساوي الصفر، لكل غرض من هذا النوع.

ويمكن أن نضيف على المرجع الأساسي، ما يمكن أن نلخصه في هذا الموضوع بما يلي :

التصريح دون إعطاء قيمة بدائية:

بفرض أنّ T1 نوع معطيات ما، وليكن Ob1 غرض ما من T1، فإنه يتم التصريح عن Ob1 على الشكل التالي:

Ob1:T1;

وبفرض أن T1 نوع معطيات ما، وليكن Ob1 وOb2 غرضين من T1، فإنه يتم التصريح عنهما كما يلي:

Ob1,Ob2:T1;

أو

Ob1:T1;
Ob2:T1;

التصريح، مع إعطاء قيمة بدائية:

بفرض أنّ T1 نوع معطيات ما، وليكن Ob1 غرض من T1 نريد التصريح عنه مع إعطائه قيمة بدائية Val1، فيتم ذلك على الشكل التالي:

Ob1 : T1 := Val1;

وبفرض أنّ T1 نوع معطيات ما، وليكن Ob1,Ob2,Ob3 أغراض منه، ونريد التصريح عن هذه لأغراض، وإعطاء قيمة بدائية للغرض Ob1 ولتكن Val1، فيتم ذلك على الشكل التالي:

Ob1 : T1 := Val1;
Ob2 : T1;
Ob3 : T1;

أو

```
Ob1 : T1 :=Val1;
Ob2, Ob3 : T1;
```

مثال :

ليكـن Aircraft غـرض مـن Aircraft_Record، ونريـد التصريح عنـه مـن أجـل kind=Military، مع إعطاء مركباته القيم البدائية التالية :

Airspeed=150.0

Heading=97.3

Latitude=147.6

Longitude=27.1

Classification=Transportd

Source=France

فيتم التصريح عن ذلك، كما يلي :

```
Aircraft : Aircraft_Record(Military) :=(Airspeed => 150.0,
                    Heading         => 97.3,
                    Latitude        => 147.6,
                    Longitude       => 27.1,
                    Classification  => Transport,
                    Source          =>    France );
```





مسألة التصميم الثانية: نظام قاعدة معطيات
Data Base System

تعريف المسألة

تحديد الأغراض

تحديد العمليات

تأسيس الرؤية

تأسيس واجهة التخاطب



لا يعتبر زرع نظام قاعدة معطيات كتطبيق نظام محمول، إذ أن نظم قواعد المعطيات، تنتمي بشكل تقليدي إلى مجال معالجة المعطيات الإدارية في COBOL، واللغات المرتبطة بها. أما لغةADA فهي ذات استخدام عام وهي مناسبة لهكذا تطبيقات. وبمعنى أصح فإن النظم الضخمة جداً، تبدو أكثر فأكثر محتاجة لممارسة قواعد المعطيات كجزء من حلولها.

ومن الفصل ٧ وحتى الفصل ١٠، سندرس بنية نظام بسيط لقاعدة معطيات. ففي الفصل ٧، سنتبع طريقتنا غرضية التوجه، لخلق وحدات برمجية لواجهات تخاطب. وبما أننا سنحتاج إلى موارد أخرى لإنهاء الحل، فاننا سندرس البرامج الجزئية والتعليمات في الفصلين ٨ و ٩، ونكمّل التنفيذ البرمجي (الزرع) لحل المسألة في الفصل ١٠.

٧ - ١ - تعريف المسألة (Define the Problem) :

تدخل قواعد المعطيات في عدد هائل من التطبيقات. فمثلاً، نحتاج إلى استخدام قاعدة معطيات لجمع معلومات حول الطلبيات أو العاملين في مؤسسة. وفي أيّة حالة، يمكن أن نمتلك نظام إدارة قاعدة معطيات نكيّفه وفق احتياجاتنا. ومع ذلك ليس من الضروري أن تشكل أنظمة قاعدة المعطيات المستخدمة من أجل تطبيق محدد، نظاماً ذا أهداف عامة. وبالتالي، يمكن أن يكون مفضلاً تطوير نظام قاعدة معطيات مخصص، ويرتبط باحتياجات المسألة المطروحة. وبالحقيقة، هذه هي الطريقة التي سنتبعها في هذا الفصل، لتعطينا فرصة لكشف كيف يمكننا تطبيق ADA لهكذا مسألة، مع إظهار تسهيلات جديدة وكثيرة في اللغة.

فتخيل الآن بأننا نملك مجمعاً ضخماً من الألبومات المسجلة. ومن فترة لأخرى، يمكننا إضافة ألبومات جديدة، أو التخلص من الألبومات التي لم نعد بحاجة لها، أو تبديل الألبومات القديمة بألبومات جديدة. وعندما يزداد عدد الألبومات، يُصبح من المجهد أكثر فأكثر أن تبحث يدوياً في مجمع الألبومات لإيجاد أغنيات خاصة، أو قطع ذات أسلوب معين، أو أغنيات لفنانين محدّدين. فمن أجل هذه الأسباب، نرغب ببناء نظام بسيط لقاعدة معطيات لمساعدتنا بالسيطرة على مجمعنا.

فما هو دفتر متطلبات نظامنا؟ بشكل أساسي، نحتاج لقاعدة معطيات وفق أسس ADA، وتحوي كافة المعلومات عن كل ألبوم في تجمعنا. ومن أجل كل

ألبوم، نعتزم الحفاظ على عنوان، واسم الفنان، وأسلوب الموسيقا (على سبيل المثال، تقليدية، جاز، أو الروك) والإصدار، واسم وطول كل أغنية. وسنحتاج للتسهيلات العادية لإدارة قواعد المعطيات، وإمكانية خلق، وفتح، وإغلاق قاعدة معطيات وإمكانية إضافة، وحذف، وتغيير معطيات كل ألبوم. وبالإضافة لذلك، نريد أن نكون قادرين على توليد تقارير عن الألبومات من قاعدة معطياتنا. وبشكل خاص، نريد أن نكون قادرين على البحث عن ألبومات وفق أي واصف. (على سبيل المثال، نرغب بإيجاد جميع الألبومات التي تمّ إصدارها في سنة خاصة). ونريد أيضاً، أن نكون قادرين على فرز التسجيلات الناتجة، ومن ثمّ طباعة تقرير. ولتقديم أعظم مرونة في بحثنا، سنسمح للمستخدم بتعين أي عدد اختياري من معايير البحث التي يجب أن تلائم كل تقرير. وبالتالي، يمكن للمستخدم البحث عن التسجيلات لفنان خاص، ومن ثمّ يبحث في تلك التسجيلات التي وجدها عن ألبومات تمّ إصدارها بسنة محددة. وبهذه الطريقة، يتم البحث عن مجموعات تسجيلات أصغر فأصغر، وليس على قاعدة المعطيات كاملة.

فمن الواضح، أننا لم نعبر عن جميع المتطلبات الضرورية لنا لفهم المسألة بشكل كامل. وعلى أية حال، لدينا معلومات كافية للمتابعة بالطريقة غرضية التوجه. وأثناء العرض، سنكشف بقية المتطلبات.

٧ ـ ٢ ـ تحديد الأغراض (Identify the Objects) :

كيف يمكننا التقدم بخلق حل برمجي لمسألتنا؟ بطريقة وظيفية، سنبدأ بتعين الخطوات الأساسية للإجراءات العامة. وعلى أية حال، مثلما لاحظنا في الفصلين ٢ و٣، غالباً ما تقود هذه الطريقة لحلول لا تتبع مجال المسألة وقابلة للتغيُّر. ولذلك سنبدأ بتعيين الأغراض، وصفوف الأغراض التي توجد في فضاء المسألة، مثلما فعلنا في الفصل ٥، من أجل الفهرسة الأبجدية.

ويحتوي وصفنا للمسألة على عدد من الأسماء لوصف الكيانات الخاصة بها. بشكل خاص، ذكرنا الكيانات:

ألبومات (Albums).

قواعد معطيات (Database).

تقارير (Reports).

ومع ذلك، توجد فروقات هامة بين هذه الكيانات. ففي قاعدة معطياتنا، يمكن أن يوجد العديد من الألبومات؛ إن ALBUMS تمثل إذن صف أغراض. ومن جهة أُخرى، فإن وصفنا لا يمثل سوى قاعدة معطيات نشطة واحدة فقط؛ وبسبب ذلك، مثلما سنناقش في الفصل ١١، سنوصف ALBUMS وكأنها نوع معطيات مجرد. وتمثل «قاعدة معطيات» (Database) آلة حالة–مجردة. ولا يوجد أيّ سبب لعدم إمكانيتنا لبناء أكثر من تقرير مرةً واحدة، لذلك سنعالج «تقارير» (Reports) كتعريف لنوع معطيات مجرد.

ومثلما ذكرنا في الفصل ٤، فكل برنامج في ADA يتطلب وجود برنامج جزئي، يعمل كجذر للنظام. وبالحقيقة، فإن الجذر الفعلي ليس غرضاً، ولكن (مثلما سنرى في مقطع لاحق) يقودنا هذا الطلب إلى إدخاله في حلنا على شكل برنامج جزئي ذي ترجمة منفصلة. وبما أنّ استدعاء هذا البرنامج الجزئي يمثل العمل الذي يستهل تطبيقنا، لذلك سنسميه بجملة اسمية نشطة، Process_Album_Database. ومثلما فعلنا للمسألة المطروحة بالفصل ٥، سيقوم البرنامج الرئيسي بمعالجة جميع طلبات المستخدم. ومن الواضح، في نظام ضخم، أنه يمكن تعليب واجهة تخاطب المستخدم–الآلة كتصميم غرضي التوجه مستقل.

٧ ـ ٣ ـ تحديد العمليات (Identify the Operations) :

لقد عينا الأغراض الأساسية الهامة، ولكن لسنا مستعدين بعد، لتأسيس بنية حلّنا. فأولاً، يجب أن نأخذ بعين الإعتبار سلوك كل تجريد.

فعلى سبيل المثال، دعنا نفحص الغرض Database. فحسب وصف مسألتنا، يحتوي هذا الغرض على مجمع من ألبومات شخصية. ونتوقع أن تكون Database قادرة على تحمل عمليات قاعدة المعطيات العادية:

Create -- make a new database

Open -- open an existing data base

Close -- close the current database

Add -- insert a new record into the database

Delete -- remove a record from the database

Modify -- alter an existing record in the database

وتشكل هذه العمليات «البنّاءات» (Constructors)، التي يمكن أن تخضع لها قاعدة معطيات. (تذكّر بأنّ البنّاءات تمثل عمليات تغيُّر في حالة غرض). وبشكل مشابه، نحتاج لمجموعة «مختارات» (selectors)، تسمح لنا بإعادة إيجاد حالة قاعدة معطيات. فعلى سبيل المثال، يمكننا تضمين المختارات التالية:

Size_Of -- return the number of records in the database

Value_Of -- return the value of one particulare record in the database

لاحظ التوازي: حيث يسمح لنا بنّاء Add بحشر تسجيلة جديدة، ويسمح لنا المختار Value_Of بإعادة إيجاد قيمة تسجيلة خاصة.

فكيف يمكننا المرور على كل تسجيلة في قاعدة المعطيات؟ يمكننا تضمين «مكرّر» Iterator. والمكرّر ليس عملية بسيطة؛ ولكنه معرّف بواسطة مجموعة من العمليات، التي تسمح لنا بالمرور على جميع العناصر في قاعدة المعطيات. وبشكل نموذجي، يتضمن المكرّر العمليات التالية:

Initialize -- establish a starting point for the traversal

Get_Next -- advance the iterator to designate another item

Value_Of -- return the record currently being designated

Is_Done --return True when all records have been visited

لاحظ بأنّ التكرار لا يعين ترتيباً خاصاً للتجوال؛ ويضمن التكرار فقط، أنه عندما ينتهي تكون قد تصفحنا جميع الأصناف.

بينما Database تشير لآلة حالة-مجردة، فإن Reports، يمثل نوع معطيات مجرد. ومن أجل البدء بتقرير، يجب أن نضمن العملية التالية:

Initialize -- start up the report so that it includes all
-- the records in the data base

وتسمح مسألتنا للمستخدم بالبحث عن ألبومات وفق أصناف خاصة. وبالتالي يجب أن نضمن البنّاء التالي:

Find -- select the albums within a given
-- that satisfy the given crite eportria

وبشكل خاص، سنسمح للمستخدم باختيار ألبومات حسب العنوان، والفنان، والأسلوب، وسنة الإصدار، وعدد التسجيلات، واسم وطول الأغنية. وبالتالي، لا يمكننا تصدير عملية Find واحدة فقط، بل يجب أن نقدم Find من أجل كل فئة.

وبما أنّ وصفنا للمسألة يتطلب Find ، فإنّ البحث وفق Find يتم في مجموعات ألبومية صغيرة. وبالتالي، عندما نطبق Initialize على تقرير، نبدأ بجميع الألبومات في قاعدة المعطيات. وكلما استدعينا Find من أجل صنف خاص، فإننا لا نحتفظ إلا بألبومات التقرير، والتي تُرضي القيمة المعطية. فعلى سبيل المثال، يمكننا استدعاء Find من أجل أسلوب ألبوم خاص، ونحدد فقط تلك المعلومات التي تخص الألبومات الكلاسكية التي يمكن الإحتفاظ بها في التقرير. ومن ثم يمكننا استدعاء Find من أجل سنة خاصة، ولا نحتفظ إلا بالمعلومات حول الألبومات الكلاسكية، التي تمّ إصدارها في تلك السنة. ومن الواضح، أنه يمكننا تأسيس مجموعة من معايير Find التي لا يمكن تطبيقها على أي ألبوم - مثل استدعاء Find من أجل السنة ١٩٣٠، وأيضاً من أجل السنة ١٩٥٠، ولذلك، وفق تعريف مسألتنا، فإننا نحتاج لبنّاء آخر:

Sort -- ordre the albums in the report according
-- to the given criteria

ويجب أن يتضمن المعيار نوع الترتيب (تنازلياً أو تصاعدياً)، والمفتاح الذي يجب استخدامه. وبشكل خاص، سنتمكن من فرز التقارير وفق العنوان، والفنان، والأسلوب، والسنة، وعدد الأغاني.

و مثلما احتجنا لمختار من أجل Database، فإننا نحتاج أيضاً لمختار ليُبلغنا عن حجم غرض تقرير:

Length_of	-- returns the number of albums the report
	-- describes

ونحتاج أيضاً لمكرِّر، لنتمكن من تصفح كافة الأنواع في التقرير. وبالتالي، يجب أن نضمن العمليات التالية:

Initial	-- establish a starting point for the traversal
Get_Next	-- advance the iterator to designate another item
Value_Of	-- return the record currently being visited
Is_Done	-- return True when all records have been visited

ومثلما سنرى في مقطع لاحق، يسمح لنا المكرِّر بطباعة معلومة عن جميع الألبومات في تقرير محدد، دون تغير حالة التقرير نفسه.

والكيان الأخير Albums، يعرِّف صف أغراض. ومع ذلك، فإننا سنستخدم طريقة لوصف سلوكه، مختلفة عن طريقتنا في Database وReports. وبالتحديد، لقد عرفنا Database

و Reports كتجريدات معلبة؛ بينما سنعبِّر عن Albums ككيان غير معلب.

ومثلما ناقشنا في الفصل ٦، فإن الأنواع الخاصة والخاصة المحدودة، تخفيا الرؤية الداخلية (التنفيذ البرمجي) التجريد. وبالتالي، تُستخدم هكذا أنواع لتعليب قرارات التصميم، وإخفائها من الرؤية الخارجية. وإن جميع الأنواع الأخرى، توفر تجريدات غير معلبة؛ إذ أنّ بنية كل نوع تكون مرئية بالكامل من الخارج. وكقاعدة عامة، إننا نتجه لاستخدام التعليب، عندما يبدو أنه من الخطر ترك الزبون يصل إلى تمثيل البنية التحتية لكيان. وعلى سبيل المثال، إذا أردنا لزبون أن يعالج التمثيل

الحقيقي لـ **Database** برمجياً، سيكون ممكناً وضع الغرض في حالة غير متماسكة. وبالمقابل، فإن تمثيل ألبوم شخصي، هو تجريد بنيوي صرف. وهذا يعني، أنّ غرض برنامجنا، يُستخدم ببساطة لتجميع أغراض أُخرى (في هذه الحالة، عنوان، فنان، سنة، إلخ..). وبالتالي، فإن هذا غير مفيدٍ بتعليب التجريد، لأنّ كل ما يستطيع فعله الزبون، لا يمكنه التأثير على حالته.

ولذلك، ففي هذه الخطوة، لن نعين صراحة أي عملية على صف الأغراض المعرّف بواسطة **Albums**. وبالتالي، سنصدر فقط بعض الأنواع الأساسية، والعمليات المسبقة التعريف الموافقة لذلك، مثلما شاهدنا في الفصل ٦.

هذا، وإننا لم نُنه بعد هذه الخطوة؛ إذ أنّه توجد مجموعة أُخرى من العمليات، يجب أن نعتبرها. فعند بناء التجريدات، نتجه لضبط سلوكها، بتعريف عمليات أساسية فقط. وتمثل العمليات الأساسية، عمليات يمكن تنفيذها برمجياً، إذا أمكننا الوصول إلى البنية التحتية للنوع. وبالتالي، **Add** يمثل بنّاء أساسي «لقاعدة المعطيات» **Database**، لأنّه يمكننا تنفيذه برمجياً، فقط، إذا امتلكنا رؤية داخلية للتجريد. ومن جهة أُخرى، طباعة تسجيلة خاصة من قاعدة المعطيات، لا تمثل عملية أساسية؛ ويمكننا تنفيذها برمجياً فقط، من وجهة نظر خارجية، لأنّ تجريدنا للألبوم، يكون غير معلبا.

وبما أنه قد تمّ تخصيص نظام قاعدة المعطيات هذا، من أجل الاستخدام الإنساني، فمن الضروري أن نوفر تسهيلات دخل/خرج نصية، لجميع التجريدات في فضاء حلّنا. فعلى سبيل المثال، نحتاج لبعض الطرق لعرض معلومات موافقة لألبوم. وبشكل مشابه، فإننا نحتاج لآلية تسمح للمستخدم بتسمية معايير البحث، ولفرزها في تقرير محدد. وبالتالي، فإنه بدلاً من تضمين هذه العمليات كجزء من توصيف كل تجريد، فقد اخترنا وضع هذه العمليات في مكان مختلف. وبهذه الطريقة، يمكننا الحفاظ على الإرتباط القوي لكل كيان. فعلى سبيل المثال، إذا ضمّنا عمليات الدخل/الخرج في التقرير في الكيان **Reports**، نكون فعليا قد مزجنا تجريدين:

التقرير نفسه وآلية الدخل/الخرج. وبالتالي، إذا غيرنا الطريقة وبالتالي طريقتنا للدخل/الخرج، فيجب أن نغير كل وحدة في حلّنا. ومن جهة أُخرى، من الممكن لنا أن نبدّل عمليات الدخل/الخرج هذه في وحدة مختلفة، لأنّ التسهيلات التي نحتاجها ليست أساسية؛ ويمكن تنفيذ هذه التسهيلات برمجياً فقط، من وجهة نظر خارجية لكل كيان.

ومن أجل كل هذه الأسباب، سنضيف لحلّنا ثلاث وحدات إضافية: Album_IO, Report_IO, Command_IO. وبالحقيقة، لا تمثل هذه الوحدات أغراضاً؛ إذ أنّ هذه الوحدات تمثل مجموعات من برامج جزئية، مثلما سنناقش في الفصل ١١. وعلى أي حال، فإن هذه الوحدات مرتبطة مع الأغراض، التي تمّ تعريفها بشكل ضئيل منذ قليل. وكل واحدة من هذه الوحدات الجديدة، تقدم خدمات غير أساسية، مبنية على قمة تجريدات أكثر أساسية. وبشكل خاص، Album_IO وReport_IO توازيان Albums وreports مباشرة. بينما Command_IO يوازي Database، ولكنا قد سميناه بشكل مختلف، ليُظهر بأنّ مجموعة العمليات هذه، تتضمن أيضاً أول مستوى تفاعل مع المستخدم (وبالتالي يجب الأخذ بالحسبان، أوامر المستخدم مثل Quit).

٧ ـ ٤ ـ تأسيس الرؤية (Establish the Visibility) :

بما أنّه حتى الآن قد وصفنا سلوك كل غرض وصف أغراض، فانه يجب أن نأخذ بعين الإعتبار، العلاقات بين هذه الأغراض . ففي هذه المسألة، فإن ربط الوحدات مع بعضها البعض هو ربطٌ مباشر. ولمتابعة هذه الخطوة، يجب أن نسأل: من أجل كل كيان، ما هي الأغراض أو صفوف الأغراض التي يجب أن تكون مرئية.

ويجب أن يكون واضحاً بأنّ Albums، هي التي تقدم التجريد المركزي في هذا الحل. ويشير Albums إلى صف الأغراض الأساسي، لمفردات فضاء مسألتنا. وبالتالي، لا يعتمد Albums على أي وحدة أُخرى، ولكن يجب على الوحدات التالية رؤية Albums، وهي:

Database

Reports

Album_IO

وبشكل مشابه، يجب على جذر حلنا، Process_Album_Database، أن يرى أيضاً Albums، لمعالجة معطيات الألبوم المُدخلة من قبل المستخدم.

وتمثل Database الكيان الأخفض مباشرة في طبقات تجريدنا. وبما أنّ Database تعتمد على Albums، فإنّ الكيانات التالية تعتمد على Database :

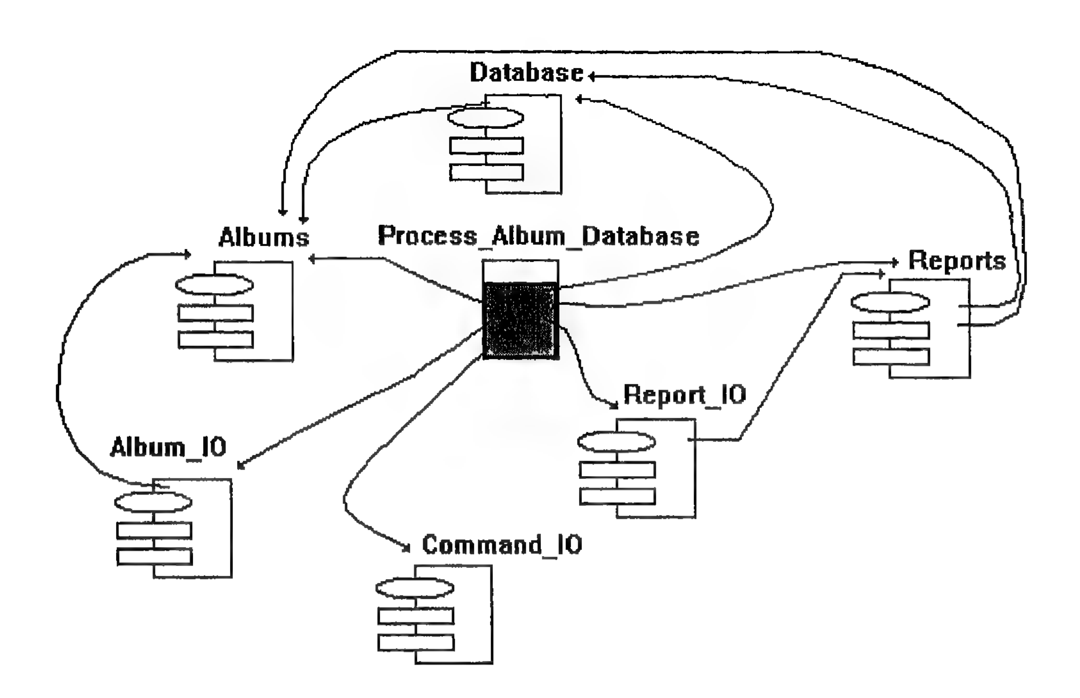
Reports

Process_Album_Database

ويجب على البرنامج الرئيسي، أن يرى معاً كلاً من Database وReports، لأنّه يمكن للمستخدم أن يتفاعل مع هذه الأغراض من خلال أوامر تعالج Process_Album_Database.

فإن الوحدتان Album_IO وReport_IO، تعتمدان على الوحدات Albums وReports، على التتالي. وبدوره، يعتمد البرنامج الرئيسي، Process_Album_Database، على هاتين الوحدتين، وأيضاً يعتمد على Command_IO لمعالجة طلبات المستخدم.

ويستخدم الشكل٧ ـ ١، الرموز التي قدمناها في الفصل٤ لتوضيح بنية نظامنا، وبالرغم من أنه في هذا الحل، يستورد البرنامج الرئيسي كل الوحدات الأُخرى، فإننا لا نجد دائماً هذه الحالة. وإنّ المسألة الحالية بسيطة نسبياً، وتتطلب هذا النوع من التفاعل. ولكن البرنامج الرئيسي للنظم الضخمة التي تمّ بناؤها في طبقات تجريد متعددة، يمكنه فقط، إدخال الوحدات ذات الطبقة الواحدة. (وعلى سبيل المثال، لاحظ الفصل ١٣، حيث نفذنا برنامجاً ذا طبقة منخفضة من المسألة التي طرحناها في الفصل٥).



الشكل ٧ ـ ١. تصميم معالجة ألبوم قاعدة معطيات Process _ Album_Database.

٧ ـ ٥ ـ تأسيس واجهة التخاطب (Establish the Interface):

الآن، وقد خلقنا التصميم عالي المستوى لحلّنا، يمكننا إدراك جميع قرارات التصميم التي صنعناها لهذه النقطة، على شكل توصيفات حزم برمجية في ADA.

ولنبدأ بواجهة التخاطب لـ Albums . ومثلما أشرنا، يتضمن الألبوم معلومات حول العنوان، والفنان، والأسلوب، وسنة الإصدار، واسم وطول الأغنية. وهذا تجريد بنيوي بسيط، يمكننا تمثيله بنوع تسجيلة. وعلى أية حال، فقد تمّ تعقيد هذا التجريد، بحقيقة أنّ كل تسجيلة يمكنها أن تحتوي عدداً مختلفاً من الأغاني. ولحسن الحظ، مثلما وصفنا في الفصل٨، تسمح لنا ADA بالتصريح عن تسجيلات ذات مميزات. وبهذه الطريقة، يمكننا كتابة التصريح التالي:

```
type Album (Number_Of_Songs : Number := 10) is
Record
    The_Title   : Title;
    The_ Artist : Artist;
    The_ Style  : Style;
    The_ Year   : Year;
    The_ Songs  : Songs(1..Number_Of_Songs);
end record;
```

وبالتالي، تتضمن التسجيلة التي تصف ألبوماً مفرداً، العنوان، والفنان، والأسلوب، وسنة الإصدار، وأيضاً مصفوفة من الأغاني، بحيث يشار إلى طولها بواسطة المميز Number_Of_Songs .

ولاحظ بأننا أسمينا كل نوع صراحة، ومثلما ناقشنا في الفصل ٦، إنّه لأكثر ضمانة أن تبني أنواعاً جديدةً مستخدماً المفردة لفضاء المسألة، بدلاً من الأنواع المسبقة التعريف.

ولإكمال واجهة التخاطب لهذه الوحدة، نحتاج معرفة طبيعة كل قطعة من معلومات الألبوم بدقة أكبر. ومن أجل أهداف هذه المسألة، سنفترض أنّ العنوان والفنان، كلاهما من النوع String بطول ٤٠. (وعلى أية حال، إن هذه السلاسل المحرفية، تمثل أنواعاً مختلفة). وسنفترض وجود مجموعة ثابتة من الأسلوب لكل ألبوم، وهكذا يمكننا تمثيلها كنوع ترقيمي. وأخيراً، نُمثل سنة الإصدار، وطول الأغنية، وعدد الأغاني بقيم رقمية. وبالحقيقة، يمكننا جعل العام ١٨٧٧ – سنة اختراع "الحاكي" بواسطة Thomas Edison – كحد أدنى. ومن جهة أُخرى، إن طول الأغنية، يمثل قيمة حقيقية، وهو يتألف من أجزاء المائة من الدقيقة.

آخذين بعين الاعتبار هذه القيود، كتجريدنا لفضاء المسألة، يمكننا ضبط تجريدنا، بالتصريحات التالية:

```
Package Albums is
   Type Title  is new String(1..40);
   Type Artist is new String(1..40);
   Type Style is (Classical, Jazz,    Rock,   Country,
                  Shows,    Religious, Ballrom, Patriotic,
                  Foreign,  Folk,      Blues,     Children);
   Type Year is range 1877..Integer'Last;
   Type Length is Delta 0.01 range 0.0..60.0;
   Type Name is new String(1..40);
   Type Song is
     Record
     The_Name : Name;
     The_Length : Length;
     End record;
   Subtype Number is private range 1..30;
   Type Songs is array (Number range <> ) of Song;
   Type Album (Number_Of_Songs : Number := 10) is
    Record
       The_Title  : Title;
       The_Artist : Artist;
       The_Style  : Style;
       The_Year   : Year;
       The_Songs : Songs(1..Number_Of_Songs);
    end record;
End Albums;
```

ويبين أسلوب الحزمة البرمجية Albums، عدة نقاط جديرة بالذكر. لاحظ أن للأنواع Title, Artist, Name البنية ذاتها، فهي تتمثل بسلاسل محرفية طولها ٤٠.

وعلى كل حال، فبالتصريح عن كل واحد كنوع وحيد، فإننا نستثمر قواعد لنمذجة الأسماء. وبالتالي، فبالرغم من أنّ هذه الأنواع متشابهة ظاهرياً، فإن قواعد لغة ADA تمنعنا من المزج بين هذه التجريدات خلال الترجمة. ولاحظ أيضاً، بأنّ نوع التسجيلة Album، يحتوي فعلياً على تصريح تسجيلة متداخلة - Song، يدخل في نوع المركب Songs، والذي هو بدوره تسجيلة. وتقود هذه البنية إلى تسمية قد تكون طويلة، ولكنها دقيقة. فعلى سبيل المثال، بإعطاء الغرض The_Album من النوع Album، يمكننا الرجوع لإسم الأغنية الثالثة في الألبوم، وفق التعبير التالي:

The_Album.The_Songs(3).The_Name

ولنتقدم الآن، من أجل دراسة الرؤية الخارجية للكيان Database . فمثلما فعلنا في الفصل ٥، يمكننا أخذ العمليات التي قمنا بتعينها سابقاً، ونعمل منها برامج جزئية مصدرة من الحزمة البرمجية. والتعقيد الوحيد في هذا الحل، يتمثل بالطريقة التي نعبر بها عن المكرّر. وتذكّر بأنّ المكرّر، يوفر آليةً تسمح للزبون بتفحص كل ألبوم في قاعدة المعطيات، دون تغيير حالته. وبالفعل، يمكن للزبون أن يرى قاعدة المعطيات، كلائحة من الألبومات. ويعمل المكرّر كدليل لهذه المجموعة. وعندما يتطور المكرّر، يمكننا ببساطة جعل الدليل يتقدم في المجموعة. طوال ذلك، يمكننا حفظ قيمة ذلك الدليل، من خلال عملية المكرّر Value_Of. وعلى أية حال، فإن هذا يحفظ فقط لمساتنا في أماكن معينةٍ من قاعدة المعطيات. وإذا أردنا إيجاد معلومة حول الألبوم في ذلك المكان، فإننا نستدعي Value_Of مختلفاً، من أجل إعادة إيجاد المعلومة. وبسبب الإرتباط الصريح لعمليات المكرّر الأربعة، فإننا سنضع هذه العمليات الأربعة في حزمة برمجية، متضمنة داخل توصيف Albums. مثلما سنرى في الفصل ١١، وهذه الطريقة أساسية لإفادة القارئ، ولا تعمل قواعد ADA إلا بقساوة زائدة علينا.

وبالتالي، يمكننا ادراك قرارات تصميمنا، في توصيف الحزمة البرمجية التالية:

with Albums;

Package Database is

```
    type Item is private;
    procedure Create (The_Name : in String);
    procedure Open   (The_Name : in String);
    procedure Close;

procedure Add     (The_Albume : in Albums.Album);
procedure Delete  (The_Albume : in Albums.Album);
procedure Modify (The_Albume : in Albums.Album;
                  To_Be      : in Albums.Album);
function Size return Natural;
function Value_Of (The_Item : in Item) return Albums.Album;
    package Iterator is
        procedure Initialize;
        procedure Get_Next;
        function Value_Of  return Item;
        function Is_Done   return Boolean;
    end Iterator;
Private
    type Item is ...
end Database
```

وبما أنّ Database تعتمد على Albums، فإنه يمكننا استيراد Albums صراحةً، باستخدام عبارة سياق.

ولاحظ بأننا قدمنا النوع Item كنوع معلب. ويُفيدنا هذا النوع، كمؤشر في لائحة الألبومات. فعلى سبيل المثال، خلال التجوال في قاعدة المعطيات، نستطيع «تذكر» بعض الأماكن. وطالما أنّ عملية المكرّر Value_Of تُعيد غرضاً من النوع Item – وليس Albums.Album – فإنه يمكن عندها التذكّر أين نحن، بوضعنا هذه القيمة في

أي غرض من النوع Item، ومن ثمّ نستدعي Value_Of مختلفة، لإعادة إيجاد معلومات الألبوم الموافقة.

وسنؤجل التنفيذ البرمجي للقسم الخاص لهذه الحزمة البرمجية، حتى الفصل١٠، حيث سندرس الرؤية الداخلية لهذا التطبيق.

وتشير Database لآلة حالة-مجردة، ولذلك لن نصدر نوع Database (هنالك قاعدة معطيات واحدة فقط). وعلى أية حال، تشير Reports لنوع معطيات مجرد، لذلك يجب أن نصدر نوع Report كنوع معلب. وبالتالي، إذا أخذنا العمليات التي عرّفناها سابقاً، يمكننا كتابة توصيف هذه الوحدة كما يلي:

```
with Albums;
package Reports is
   type report is limited private;
   type Category is (Title, Artist, Style, Year,
                     Number, Song, Length);
      subtype Sort_Category is Category range Title..Number;
type Order is (Ascending, Descending);
procedure Initialize (The_Report : in out Report);
procedure Find (The_Title  : in      Albums. Title;
                In_The_Report   : in out Report);
procedure Find (The_Title  : in      Albums. Artist;
                In_The_Report   : in out Report);
procedure Find (The_Title  : in      Albums. Style;
                In_The_Report   : in out Report);
procedure Find (The_Title  : in      Albums. Year;
                In_The_Report   : in out Report);
procedure Find (The_Title  : in      Albums. Number;
                In_The_Report   : in out Report);
procedure Find (The_Title  : in      Albums. Name;
                In_The_Report  : in out Report);
```

```
procedure Find (The_Title  : in    Albums. Length;
                In_The_Report   : in out Report);
procedure Sort (The_Report : in out Report;
                By_Category    : in   Sort_Category;
                With_The_Order : in Order := Ascending);
function Length_Of (The_Report : in Report)
                return Natural;
type Iterator is limited Private;
procedure Initialize (The_Iterator : in out Iterator;
                With_The_Report : in Report);
procedure Get_Next (The_Iterator : in out Iterator);
function Value_Of (The_Iterator : in Iterator)
                return Albums.Album;
function Is_Done (The_Iterator : in Iterator)
                return Boolean;
Private
   type Report   is ...
   type Iterator  is ...
end Reports;
```

لاحظ بأننا قد قدمنا الأنواع المرقمة Category, Sort_Category, Order بشكل أساسي لدعم واجهة التخاطب للبناء Sort. أيضاً، لقد قصدنا إجراء التحميل الزائد لقسم Find من أجل تطبيقه على جميع أصناف الفرز، لأنّ كل واحد يتطلب معاملاً من نوع مختلف.

سيدرك القارئ الحريص بأنّ مكرر Database مختلف بشكل طفيف عن مكرر Reports بما أنّ Database تمثل آلة حالة-مجردة، فلدينا غرض واحد فقط؛ لهذا السبب، نتطلب مكرراً واحداً فقط في وقت واحد. بما أن Reports يوفر نوع المعطيات المجرد Report، لذلك يجب أن نكون قادرين على امتلاك عدة مكررات. من أجل هذا السبب، قدمنا النوع مكرر، الذي يمثل معاملاً لعمليات المكرر الأربعة. نربط مكرر

(Iterator) إلى غرض تقرير خاص في العملية Initialize . دلالة هـذه العمليـة تجعـل غرض المكرر مشيراً إلى جميع الحدود في قاعدة المعطيات المحددة.

الوحـدات الثـلاث الباقيـات – Command_IO, Reports_IO, Album_IO – تُستخدم فقط كمجموعات من برامج جزئية. على سبيل المثال، حسب وصفنا السـابق، يوفر Album_IO تسهيلات دخل/خرج لكل قطعة من معلومات الألبوم:

```
With Albums;
Package Album_IO is
  Procedure Get (The_Title     : out Albums.Title );
  Procedure Get (The_Artist    : out Albums.Artist);
  Procedure Get (The_Style     : out Albums.Style );
  Procedure Get (The_Year      : out Albums.Year );
  Procedure Get (The_Number    : out Albums.Number);
  Procedure Get (The_Song      : out Albums.Song);
  Procedure Get (The_Length    : out Albums.Length);
  Procedure Get (The_Album     : out Albums.Album );
  Procedure Put (The_Album     : in Albums.Album );
End Album_IO;
```

العملية المعقدة الوحيدة في هذه الحزمـة البرمجيـة تكـون Get مـن أجـل ألبـوم كامل؛ في هذه العملية يمكننا إدخال عدد مختلف من الأغاني لكل ألبوم. نفس الشكل يأخذه Report_IO. يجـب أن نوفـر دخـل/خـرج مـن أجـل واصفـات عديـدة لتقريـر، متضمنين صنف إيجاد، صنف فرز، وترتيب فرز:

```
with Reports;
package Report_IO is
   Procedure Get(The_Category : out Reports.Category);
   Procedure Get_Sort (The_Category :
                 out Reports.Sort_Category);
```

```
    Procedure Get(The_Order  : out Reports.Order);
    Procedure Put(The_Length : in Natural);
  end Report_IO;
```

لاحظ بأنّه يجب أن نستخدم الاسم Get_Sort بدلاً من استخدام الاسم المحمل زائداً Sort بينما الاسم Sort_Category يكون فقط نوعاً جزئياً من Category . مثلما سندرس أكثر من ذلك في الفصل القادم، قواعد لغة ADA مثل قواعد التنفيذ البرمجي لا يمكنها التمييز بين استدعاء Sort من أجل النوع واستدعاء Sort من أجل النوع الجزئي؛ لذلك، يجب أن نستخدم أسماء مختلفة لتجنب الالتباس.

آخر حزمة برمجية هامة تتمثل بـ Command_IO. نلتقط هنا قرارات تصميمنا حول الطرق التي تمكن المستخدم من التفاعل مع النظام. يمكننا التعبير عن هذا الصف من الأوامر بنوع مرقم وحيد وعملية واحدة تحصل على قيمة من المستخدم:

```
 package Command_IO is
    type Command is (Create, Open, Close, Add, Delete,
                     Modify, Start_Report, Find, Sort,
                     Display_Report,
                 Quit);
    procedure Get (The_Command : out Command);
 end Command_IO;
```

لقد عيّنا الآن واجهة تخاطب لكل الوحدات في هذا المستوى من التجريد في فضاء حلنا. على أي حال، لم ندرس بعد التسهيلات التي تسمح لنا بالتعبير الكامل عن التنفيذ البرمجي. وبالتالي، سنؤجل إكمال حلنا حتى الفصل ١٠. في الفصل ٨، سنناقش البرامج الجزئية، وفي الفصل ٩، سنعتبر التعليمات كوسيلة للتحكم الخوارزمي. بالتسلح بهذه الموارد الجديدة، سنكمل حلنا لنظام قاعدة المعطيات.



8

البرامج الجزئية
Subprograms

شكل البرامج الجزئية

استدعاء البرامج الجزئية

تطبيقات البرامج الجزئية في ADA



مثلما لاحظنا سابقاً، إن ADA تحتوي العديد من الأنواع الأساسية، وكذلك آلية لخلق أنواع معطيات مجردة (باستخدام الأنواع الخاصة). والبرامج الجزئية في ADA توازي هذه المفاهيم، وذلك بآلية خلق عمليات مجردة.

٨ ـ ١ ـ شكل البرامج الجزئية

(The Form of ADA Subprograms) :

تعتبر البرامج الجزئية وحدة التنفيذ الأساسية في نظم ADA، ويمكن أن تكون من أحد الصنفين التاليين:

- **الإجرائيات (Procedures).**
- **التوابع الفرعية (Functions).**

والبرامج الجزئية، تشبه بقية الوحدات البرمجية في ADA (الحزم البرمجية، والمهمات)، حيث يمكن تقسيمها إلى قسمين أساسيين، وهما «الجسم» (Body) و«التوصيف» (Specification).

والتوصيف يحدد إسم البرنامج الجزئي، وتوضع الوسائط (المتغيرات)، وأنواعها فيه. ويمكن أن يعيد قيمةً من نوع معين، وذلك في حالة التابع الفرعي، وهو يمثل واجهة التخاطب مع الوحدة.

وفيما يلي، بعض الأمثلة عن توصيف بعض البرامج الجزئية:

```
procedure count_leaves_on_Binary_tree;
procedure push (element : in integer; On : in out Buffer);
function cos(Angle : in radians) return float;
function "*" (Left,right : in matrix) return matrix;
```

والجسم «يُغلّب» (Encapsulates) سلسلةً من التعليمات، التي تُعرِّف الخوارزمية.

الشكل العام للبرنامج الجزئي:

بما أنه يوجد صنفان للبرامج الجزئية، فلكل صنف شكل عام خاص به، مختلف بعض الشيء عن الشكل العام للصنف الآخر.

الشكل العام للإجرائية :

إن الشكل العام للإجرائية، كما يلي:

```
Procedure Procedure_Name (Parameters_List) is
-- Declaration_Part
Begin
  -- sequence of statements
End Procedure_Name;
```

– Procedure_Name «إسم الإجرائية»، يجب أن يكون مثل أيّ إسمٍ في لغة ADA.

– Parameters_List: «لائحة المتغيرات»، تمثل أغراض دخل وخرج الإجرائية، يمكن لبعض المتغيرات أن تعطي قيماً بدائية.

– Declaration_Part يمثل «قسم التصريحات»، ويتم ضمنه التصريح عن أغراض متغيرة، تساعد في تنفيذ خوارزمية الإجرائية، كما يمكن التصريح عن برامج جزئية، خاصة بهذه الإجرائية.

– sequence of statements: «سلسلة تعليمات»، تمثل جسم الإجرائية.

مثال عن الإجرائية:

فيمايلي، إجرائية تنفذ عملية حسابية على عددين صحيحين، وذلك وفق رمز خاص بالعملية.

```
procedure Arith_Operation(first : in integer;
                 second_result : in out integer;
                 operation:character;Error : out Boolean) is
 Begin
  Error := true;
  case operation is
   when '+' => second_result:=first+second_result;
   when '-' => second_result:=first-second_result;
   when '*' => second_result:=first*second_result;
```

```
when '/' | '%' =>
 if second_result=0 then
  put_line("Division By Zero !!");
  Error := false;
 else
  if operation='/' then
   second_result:=first / second_result;
  else
   second_result:=first rem second_result;
  end if;
 end if;
when others=> put_line(" Illigal Operation ! ");
 Error := false;
end case;
End Arith_Operation;
```

فوفق هذا المثال نجد أنّ:

Procedure_Name = Arith_Operation.

- Parameters_List تتمثل بما يلي:

first : in integer; second_result : in out integer;
Operation : in character ; Error : out Boolean

- Declaration_Part لا يحتوي شيئاً (فارغ).

- sequence of statements تتمثل بسلسلة التعليمات المحصورة بين Begin و End.

شرح الإجرائية:

وفق هذه الإجرائية، يتم حساب ناتج تطبيق العملية الحسابية Operation على first وsecond_result، وذلك في حال كون Operation تمثل إحدى العمليات الحسابية المعروفة، وإمكانية تطبيقها على هذين العددين. وفي هذه الحالة، نجد أنّ النتيجة

تتوضع في المتغير second_result، وإن قيمة المتغير Error تأخذ القيمة true لتشير إلى صحة العملية، وإمكانية تطبيقها، أما في غير ذلك، فستظهر رسالة تحدد سبب الخطأ، وError سوف تأخذ القيمة false، دون تغيير لقيمة المتغير second_result.

ملاحظة:

إن الحدود:

- in.
- out.
- in out.

تحدد جهة تدفق المعطيات الخاصة بالإجرائيّة، حيث أنّ:

- in تشير إلى أن الغرض الموافق، لا يمكن تغيير قيمته أثناء استدعاء الإجرائية (دخل).
- out تشير إلى أن الغرض الموافق، سيكون خرجاً للإجرائية الموافقة.
- in out تشير إلى أنّ الغرض الموافق، سيكون دخلاً وخرجاً بنفس اللحظة للإجرائية.

وفي حال عدم ذكر إحداها قبل نوع المتغير، في هذه الحالة، تعتبر جهة تدفق المعطيات من النمط in.

ولكن من المفضل ذكر نمط جهة تدفق المعطيات، وذلك لجمالية قراءة البرنامج الجزئي.

إستدعاء الإجرائية:

يتم استدعاء الإجرائية، على الشكل التالي:

Procedure_Name(Parameters_List);

يجب أن يكون ترتيب المتغيرات ضمن Parameters_List مثلما ورد في الترتيب عند توصيف الإجرائية، دون ذكر نوع كل متغير، إذ يفصل بين كل متغيرين المحرف "،"

وفي حال إعطاء قيم بدائية لبعض المتغيرات عند توصيف الإجرائية، وعدم ذكر هذه المتغيرات عند طلب الإجرائية، فإنّ هذه المتغيرات ستأخذ نفس القيم التي حددت في البداية.

مثال: يتم استدعاء الإجرائية Arith_Operation كما يلي:

```
Arith_Operation(f,s_r,Op,Er);
```

ففي هذه الحالة، سيتم تطبيق الإجرائية على f,s_r، وذلك وفقاً للعملية Op. وإشارة تحقيق العملية ستخزن في Er. وقبل استدعاء هذه الإجرائية، يجب التصريح عن:

- f وs_r بأنهما متغيرات من «الأعداد الصحيحة» (f,s_r : integer;).
- Op بأنه متغير «محرفي» (Op : character;).
- Er بأنه متغير «منطقي» (Er : Boolean;).

الشكل العام للتابع الفرعي:

إن الشكل العام للتابع الفرعي كما يلي:

```
Function Function_Name(Parameters_List)return Return_Type is
   --- Declaration_Part
Begin
   --- sequence of statements
End Function_Name;
```

- Function_Name: إسم التابع الفرعي.
- Parameters_List: لائحة المتغيرات، وتمثل أغراض دخل التابع الفرعي.
- Return_Type: تمثل نوع نتيجة التابع الفرعي Function_Name.
- Declaration_Part: يمثل قسم التصريحات، وضمنه يتم التصريح عن أغراض متغيرة، تساعد في تنفيذ خوارزمية التابع الفرعي، كما يمكن التصريح عن برامج جزئية، خاصة بهذه الإجرائية.
- sequence of statements: سلسلة تعليمات تمثل جسم التابع الفرعي.

مثال: سنورد فيما يلي تابعاً فرعياً، يحسب العاملي لعدد صحيح موجب (natural):

```
function fact(I: in natural :=7) return natural is
   k : natural :=1;
Begin
  if I>1 then
  for j in 1.. I
    loop
      k:=k*j;
    end loop;
  end if;
    return k;
End fact;
```

ووفق هذا المثال، نجد أنّ:

- Function_Name=fact
- Parameters_List=(I: in natural :=7)
- Return_Type=natural
- Declaration_Part=(k : integer)
- sequence of statements تمثل مجموعة التعليمات المحصورة بين Begin و End.

شرح بنية التابع الفرعي Fact:

نريد حساب العاملي للعدد الصحيح الموجب I، الذي ناتجه سيكون من النوع الصحيح الموجب أيضاً، ولذلك، تمّ تحديد نوع الناتج بـ return natural ؛ بعد ذلك، تمّ إعطاء قيمة بداثية للمتغير، الذي هو من النوع الصحيح الموجب، ثم تمّ حساب قيمة العاملي وفق الخوارزمية المعروفة، ومن ثم return k تشير إلى إعادة قيمة k كنتيجة لـ fact بعد انتهاء الحساب. وعند استدعاء التابع الفرعي، إذا لم تحدد قيمة المتغير ، I سيتم حساب «العاملي» للعدد ٧.

استدعاء التابع الفرعي:

يتم استدعاء التابع الفرعي بلغة ADA، مثل استدعائه في بقية لغات البرمجة عالية المستوى، مثل باسكال و C وذلك، وفق الأشكال التالية:

ـ الإسناد :

يتم إسناد التابع الفرعي F_N إلى غرض متغير O_V نوعه من نوع التابع الفرعي كمايلي: O_V := F_N(Parameters)

حيث Parameters تمثل مجموعة معاملات دخل التابع الفرعي مرتبة وفق وورود ترتيبها في قسم التوصيف الخاص بالتابع الفرعي المحدد دون تحديد نوعها، إذ يفصل بين كل اثنين المحرف "," .

ـ الشروط:

في حال كون التابع الفرعي من النوع المنطقي، فإنه يتم استدعائه وفق:

– تعليمة if كما يلي:

```
if F_N(Parameters) then
sequence of statements
end if;
```

أو كما يلي:

```
if Not F_N(Parameters) then
  sequence of statements
end if;
```

– تعليمة while كما يلي:

```
while F_N(Parameters) loop
  sequence of statements
end loop;
```

أو كما يلي:

```
while Not F_N(Parameters) loop
  sequence of statements
end loop;
```

- **الإظهار:**

عند إظهار قيمة تابع فرعي نوعه بسيط، فإنه يتم ذلك كما يلي:

```
put  F_N(Parameters) ;
```

وفي هذه الحالة، يجب أن يكون من الممكن إظهار قيمة التابع F_N وفق تابع الإظهار put.

- **تعليمة Case** : عندما تكون قيمة التابع F_N من النوع البسيط، ومن السهل مقارنتها مع عدة قيم وفق تعليمة Case، يتم ذلك كمايلي:

```
Case F_N(Parameters) is
  Where V1 => Statements1
  Where V2 => Statements2


  Where Vn => Statementsn
end Case;
```

مثال: فيما يلي، برنامج بسيط يتم وفقه استدعاء التابع Fact بعدة طرق مختلفة:

```
WITH Text_IO; USE Text_IO;
PROCEDURE facto IS
package nat_io is new integer_io(natural);use nat_io;
fact3:natural;
function fact(i: in natural :=7) return natural is
 k: natural :=1;
begin
 if i>1 then
  for j in 2..i
  loop
   k:=k*j;
  end loop;
```

```
 end if;
 return k;
end fact;

BEGIN
 fact3:=fact(3);
 put("Fact(5)=");put(fact(5));new_line;  --- fact(5)=120
 put("fact(7)=");put(fact);new_line;  --- fact=fact(7)=5040
 put("fact(3)=");put(fact3);new_line;  --- fact3=fact(3)=6
END facto;
```

ملاحظة: بما أنّ أنواع الوسائط (parameters) في التوابع الفرعية جميعها من الشكل in، فليس من الضروري ذكر in قبل نوعكل متغير.

التحميل الزائد (Overloading):

في بعض الأحيان، يحتاج المبرمج لإعطاء اكثر من برنامج جزئي، نفس الإسم، ضمن الوحدة البرمجية الواحدة أو ضمن البرنامج الواحد. ولكن أكثر لغات البرمجة العالية المستوى، لم تسمح بذلك، بينما سمحت لغة ADA بذلك، ولكن ضمن قيود. وتتمثل هذه القيود بعدم تكرار نفس الترتيب والعدد من متغيرات دخل/خرج الإجرائية ومن نفس النوع أكثر من مرة من أجل اسم واحد. وتُدعى هذه العملية بالتحميل الزائد:

وفيما يلي، برنامج بسيط يحتوي ثلاث إجرائيات، بنفس الإسم من أجل متغيرات دخل/خرج مختلفة الأنواع لكل إجرائية:

```
WITH Text_IO; USE Text_IO;
PROCEDURE arit_op IS
f1,f2,f_r:float;
i1,i2,i_r:integer;
procedure add(f1,f2: in float; f_r : out float) is
begin
 f_r:=f1+f2;
```

```
end add;
procedure add(i1,i2 : in integer; i_r : out integer) is
begin
 i_r:=i1+i2;
end add;
procedure add(i1:in integer;f1:float;i_r : out integer) is
begin
 i_r:=i1+integer(f1);
end add;
package fl_io is new float_io(float);use fl_io;
package int_io is new integer_io(integer);use int_io;
BEGIN
 put("Enter 2 Integers: ");
 get(i1);get(i2);
 put("Enter 2 Floats: ");
 get(f1);get(f2);
 add(i1,i2,i_r);
 put_line("i1+i2=" & integer'image(i_r));
 add(f1,f2,f_r);
 put("f1+f2=");put(f_r);new_line;
 add(i1,fr,i_r);
 put_line("i1+integer(f_r)=" & integer'image(i_r));
END arit_op;
```

شرح البرنامج:

لقد تمّ في هذا البرنامج تعريف ثلاث إجرائيات تحت الاسم add، ولكل إجرائية ثلاث متحولات دخل/خرج من مختلف الأنواع، وبالتالي، لا يوجد التباس في ذلك، إذ أنّ متغيرات (وسائط الإجرائيّة):

- الأولى هما f1,f2 من النوع float وهما عبارة عن دخل، وكذلك f_r وهو خرج حقيقي، يمثل ناتج جمع f1 إلى f2.

– والثانية هما i1,i2 من النوع integer وهما عبارة عن دخل، وكذلك i_r وهو خرج صحيح، يمثل ناتج جمع i1 إلى i2.

– والثالثة هما i1 من النوع الصحيح وهو دخل، وf1 من النوع الحقيقي، أما الناتج i_r وهو خرج صحيح، يمثل ناتج جمع القسم الصحيح من f1 إلى i1 .

ملاحظة: يمكن تعريف إجرائية add من أجل وصل سلسلتين محرفيتين مع بعضهما البعض.

٨ ـ ٢ ـ إستدعاء البرامج الجزئية (Subprogram Calls) :

طريقة تنشيط برنامج جزئي:

نفرض وجود عدة برامج جزئية، لها التوصيفات التالية:

```
procedure Search_File(Key : in Name;
 Index  : out File_Index);
Procedure Sleep        (Time   : in Duration :=10.0);
procedure Sort         (Data   : in out Names;
              Order  : in Direction := Ascending);
procedure Sort          (Data   : in out Numbers;
              Order : in Direction := Ascending);
Procedure Turn_of (Light  L: in Location );
```

لاحظ أنّه يوجد تحميل زائد من أجل الإجرائية Sort، ولا يوجد التباس بسبب أنّ نوع المعطيات مختلف في المرتين.

أيضاً لاحظ، أنّه من أجل بعض الأغراض المتغيرة، قد تمّ إعطاؤها قيماً بدائية، مثل Time في الإجرائية Sleep، حيث قيمتها البدائية 10.0.

فكيف يمكن تنشيط أيٍ من مجموعة البرامج الجزئية هذه، بلغة ADA ؟

ففي لغة ADA، توجد ثلاث طرق لتنشيط البرامج الجزئية، وهي التالية:

- التمثيل (التدوين) الموضعي (Positional Notation):

وهذه الطريقة، تستخدمها أكثر لغات البرمجة عالية المستوى، مثل باسكال وC و فورتران...

وفي التمثيل الموضعي، يجب أن يتوافق ترتيب المتغيرات الفعلية، مع المتغيرات الصورية.

أمثلة:

```
Search_File("Smith, J",Record_Entry);
Sleep(120.0);
Sort(Personnel_Names,Descending);
Sort(Grades,Ascending);
```

فإذا لم يتوافق النوع الثابت لأحد المتغيرات الفعلية، مع الأنواع الموافقة من المتغيرات الصورية، سيكون هناك خطأ أثناء ترجمة البرنامج، وسوف يشار إليه.

- مجموعة المتغيرات المسمّاة (Named Parameter Association):

وتستخدم هذه الطريقة، لجعل البرامج الجزئية المستدعاة، مقروءةً بشكل أفضل.

أمثلة:

```
Search_File(Key =< "Smith, J" , Index => Record_Entry);
Sleep(Time => 120.0);
Sort(Data => Personnel_Names , Order => Descending);
```

ملاحظات:

- في حال تسمية أحد المتغيرات، يجب تسمية بقية المتغيرات.
- عند استخدام هذه الطريقة، ليس من الضروري أن يكون ترتيب المتغيرات الفعلية، هو نفس ترتيب المتغيرات الصورية، وليس من العملي تحقيق ذلك.

- الإستدعاء باستخدام المتغيرات ذات القيم البدائية:

(Calling Subprogram Using Default Parameters):

أمثلة:

يمكن استدعاء Sort، على الشكل التالي:

Sort(Personnel_Names);
Sort(Grades);

وفي كلتا الحالتين، تمّ اعتبار Direction=Ascending

أما Sleep، فيمكن استدعاؤه على الشكل التالي: ;Sleep

ففي هذه الحالة، تعتبر قيمة Time مساوية لـ 10.0 .

ملاحظات:

- إنّ استخدام هذه الطريقة، يجعل البرامج أقل قابليةً للقراءة.
- يمكن دمج هذه الطريقة، مع طريقة مجموعة المتغيرات المسمّاة.
- مثال : `Sort(Data=>Personnel_Names)`

ملاحظة: إن الطرق الثلاث السابقة الذكر، يمكن تطبيقها على الإجرائيات والتوابع الفرعية، ولكن هنالك طريقة أخرى من أجل التوابع الفرعية وهي (Infix Notation). مثال ذلك، ليكن لدينا توصيف تابع فرعي يقوم بجمع مصفوفتين كمايلي:

Function "+" (Left,Right : in Matrix) return Matrix;

ففي هذه الحالة، يمكن استدعاء هذا التابع الوظيفي وفق هذه الطريقة، كمايلي:

Sum:=First_Matrix + Second_Matrix;

و ذلك بفرض أنّ Sum, First_Matrix,Second_Matrix من النوع Matrix الذي نفترض أنّه معرّف. ولاحظ أنه يوجد تحميل زائد للتابع «+»، الذي تمّ تعريفه من أجل المصفوفات، مع تابع الجمع المعرّف على مختلف أنواع الأعداد.

والتوابع الفرعية والإجرائيات، يمكن التصريح عنها دون أي متغير. وغالباً نصرح عن بعض التوابع الفرعية، عندما نحتاج للتعبير عن بعض الإسناديات (predicate)، أمثلة ذلك ما يلي:

```
Function Is_Valid_Operation return Boolean;
Function Access_Authorized return Boolean;
```

كما يمكن استدعاء بعض التوابع الفرعية دون متغيرات حقيقية. وأمثلة ذلك مايلي:

```
If Access_Authorized Then
   --- Statements
End If;
```

العودية (Recursive):

إن لغة ADA، مثل أكثر اللغات عالية المستوى، يمكن أن تستخدم العودية. والعودية، هي أن يستدعي البرنامج الجزئي نفسه، أو أن يستدعي برنامجاً جزئياً يستدعيه.

فكيف يمكن أن يتم ذلك بلغة ADA؟

فيمايلي مثال بلغة ADA عن برنامج جزئي (تابع فرعي) لحساب «العاملي» لعدد صحيح موجب، وذلك باستخدام العودية:

```
Function Factorial(I : Natural) return Natural is
Begin
  if I<2 then
    return 1;
   else
     return I*Factorial(I-1);
  end if;
 End Factorial;
```

٨ ـ ٣ ـ تطبيقات البرامج الجزئية

subprograms ADA Application for:

للبرامج الجزئية بلغة ADA ثلاث تطبيقات أولية ، وهي مايلي :

- وحدات البرنامج الرئيسي.
- تعريف التحكم الوظيفي.
- تعريف عمليات على أنواع المعطيات المجردة.

وسنناقش فيما يلي ، ونذكر أمثلة عن كل تطبيق أولي من تطبيقات البرامج الجزئية.

البرامج الجزئية كوحدة البرنامج الرئيسي (Main Program):

لم تحتو ADA بنيةً منفصلةً من أجل البرنامج الرئيسي ، إذ يمكن أن يُستخدم البرنامج الجزئي ، كوحدة لبرنامج رئيسي ، يمكن أن تنفذ دون غيرها من بقية الوحدات.

والشكل العام للبرنامج الرئيسي :

```
With Package1,Package2,...,Package_n;
Procedure  Program_Name Is
 Declaration_Part
Begin
  ---Statements
End Program_Name;
```

إذ أنّ Package1,Package2,...,Package_n عدة حزم برمجية ، كل واحدة منها تحتوي مجموعةً من البرامج الجزئية ، لتؤدي غاية محددة ضمن البرنامج الرئيسي Program_Name ، أما Declaration_Part ، فيمثل قسم التصريحات الخاصة بالبرنامج الرئيسي Program_Name ، أما Statements فتمثل مجموعة «التعليمات» الخاصة بالبرنامج الرئيسي Program_Name.

مثال: فيما يلي برنامج رئيسي، يتم وفقه حساب العاملي للعددين الصحيحين الموجبين ٥ و ٦، وذلك باستدعاء تابع عودي خاص بهذا البرنامج وهو Factorial:

```
WITH Text_IO; USE Text_IO;
PROCEDURE fact IS
package nat_io is new integer_io(natural);
use nat_io;
function factorial(i:natural) return natural is
begin
if i<2 then
 return 1;
 else
 return i*factorial(i-1);
end if;
end factorial;

begin
put("5!=");
put(factorial(5));
new_line;
PUT("6!=");
put(factorial(6));
new_line;
end fact;
```

ووفق هذا البرنامج، نجد أنّ:

- الكتل البرمجية هي فقط Text_Io.
- Program_Name=fact .
- Declaration_part يمثل مايلي:
- package nat_io is new integer_io(natural); .
- Use nat_io; .
- والتابع الفرعي Factorial .
- Statements تتمثل بمجموعة التعليمات المحصورة بين BEGIN و END fact;.

تعريف التحكم الوظيفي (Functional Control):

في التصميم الوظيفي التقليدي التنازلي Top-Down، سيفحص المحلل المسألة ابتداءً من المستويات العليا، ومن ثم يقسم الحل إلى عدة وظائف أساسية. وهذا العمل موازٍ لطرق التصميم غرضية التوجه لتجريد المعطيات.

وبعد تحديد التوابع الفرعية، يتم خلق البرامج الجزئية التي تنفذ هذه الأوامر. فإذا كانت هنالك قيمة وحيدة يجب أن تعاد، يُحبذ استخدام التابع الفرعي من أجل ذلك، وإلا فيُحبذ استخدام الإجرائية.

وبما أنّ هذه البرامج الجزئية تنفذ عملاً معيناً، فإنه يجب تسميتها بجملةٍ ذات معنى. ومثال ذلك Check_Limits Initiate_Process, Retract_Probe.

ومن المفضل أن تسمى التوابع الفرعية كأسماء، إذا كانت تعيد قيمة غرض بسيط. ومثال ذلك Cos, Random, Sensor_Value ، أما إذا كانت تعيد قيمةً منطقية، ناتجةً عن فحص شرطٍ معين، فيجب أن تسمى على شكل فعل الكون. مثال : Process_Is_Terminated, Probe_Is_Down.

وفي أي مستوى من برنامج ما، فقط البرامج الجزئية الضرورية للحل، يجب أن تكون مرئية. وبالطبع في المستويات الدنيا، يمكن أن تتطلب برامج فرعية لتنفيذ عملياتٍ عالية، ولكن هذه البرامج الجزئية غير مرئية.

مثال:

فيما يلي مثال يحتوي ثلاث إجرائيات، وهما على الترتيب:

- Read_Array: وفق هذه الإجرائية تتم قراءة عناصر مصفوفة من الأعداد الصحيحة.
- Write_Array : وفق هذه الإجرائية تتم كتابة عناصر مصفوفة من الأعداد الصحيحة.
- Inverse_Array: وفق هذه الإجرائية يتم نسخ معكوس مصفوفة من الأعداد الصحيحة، وتخزين الناتج في شعاع جديد.

لاحظ أنّه في البدء، تمّت كتابة توصيف كل إجرائية وبجانبها تمّت كتابة Separate، وهذا يعني أنّ كلاً من هذه الإجرائيات مرئية بالنسبة للبرنامج، وأنّه سيُترجم جسم كل من هذه الإجرائيات، بشكل منفصل عن البرنامج الرئيسي.

أيضاً لاحظ، أنّه قبل بدء كتابة جسم البرنامج الجزئي، تمّ تحديده إلى أي برنامج رئيسي، وذلك بوضع Separate(Program_Name) كمايلي:

```
Separate(Array_Operation)
Procedure Read_Array(Array_Length : In Integer;
              Out_Array    : Out Array_Int) Is
Begin
 for i in 1..Array_Length loop
  get(Out_Array(i));
 end loop;
End Read_Array;
```

ففي هذه الحالة Program_Name=Array_Operation، أي أنّ هذه الإجرائية تابعة للبرنامج الرئيسي Array_Operation.

والبرنامج التالي يوضح ذلك.

```
WITH Text_IO; USE Text_IO;
PROCEDURE Array_Operation IS
Max_Length :Constant Integer :=10;
Type Array_Int Is Array(1..Max_Length) Of Integer;
Array1,Array2:Array_Int;
Array_Length: Integer;
Package Int_IO Is New Integer_IO(Integer);
Use Int_IO;

Procedure Read_Array   (Array_Length : In Integer;
                        Out_Array    : Out Array_Int) Is Separate;
Procedure Write_Array  (Array_Length :In Integer;
                        In_Array     : In Array_Int) Is Separate;
```

```
Procedure Inverse_Array(Array_Length : In Integer;
                        In_Array : In Array_Int;
                        Out_Array : Out Array_Int) Is Separate;
BEGIN
 Loop
 put("Enter The Dimension Of The Table: ");
 get(Array_Length);
 Exit when Array_Length<1;
 if Array_Length>Max_Length then
  Put_Line("Error: " & Integer'image(Array_Length) & ">"
& integer'image(Max_Length));
  else
   Read_Array(Array_Length,array1);
   put("The Elements Of The Array Are : ");
   Write_Array(Array_Length,array1);
   Inverse_Array(Array_Length,Array1,Array2);
   put("The Elements Of The Inverse Are : ");
   Write_Array(Array_Length,Array2);
 end if;
 End Loop;
 put_Line("-----  The End ..   -----");

END Array_Operation;

Separate(Array_Operation)
Procedure Read_Array(Array_Length : In Integer;
                     Out_Array    : Out Array_Int) Is
Begin
 for i in 1..Array_Length loop
  get(Out_Array(i));
 end loop;
End Read_Array;
-------------------------
```

```
Separate(Array_Operation)
Procedure Write_Array(Array_Length : In Integer;
              In_Array     : In Array_Int) Is
Begin
 for i in 1..Array_Length loop
  put(In_Array(i));
 end loop;
 new_line;
End write_array;
-----------------------------
Separate (Array_Operation)
Procedure Inverse_Array(Array_Length : In Integer;
               In_Array     : In Array_Int;
               Out_Array    : Out Array_Int) Is
Begin
 for i in 1..Array_Length Loop
  Out_Array(i):=In_Array(Array_Length+1-i);
 end Loop;
End Inverse_Array;
```

وليس من الضروري أن تتواجد الإجرائيات والبرنامج الرئيسي بملف واحد، بل يجب أن تترجم، قبل تنفيذ البرنامج الرئيسي.

وإنّ استخدام البرامج الجزئية بهذه الطريقة، يجعل البرامج أكثر قابليةً للقراءة والصيانة.

وعلى أية حال، في مجال الزمن الحقيقي، إن استدعاء عددٍ ضخم من البرامج الجزئية، يسبب زمن تراكم واضح عند التنفيذ. ولإزالة هذا التراكم، مع المحافظة على بقية فوائد وحدوية البرامج الجزئية، يمكن أن يتم ذلك باستخدام موجه المترجم Pragma باستدعاء Inline. وهذا الموجه، يسمح بجلب جسم البرنامج الجزئي، ووضعه في النقطة حيث يتم استدعاؤه بها. ومن أجل تحقيق ذلك، يجب أن يظهر موجه

المترجم Inline في نفس قسم التصريحات، حيث يتم التصريح عن البرامج الفرعية، ويتم ترميزه على الشكل التالي:

لاحظ أنّ وسائط موجه المترجم Inline عبارة عن أسماء البرامج الفرعية .

بالنسبة للمثال السابق يجب إضافة السطر:

```
Pragma Inline(Read_Array, Write_Array, Inverse_Array);
```

قبل السطر:

```
Procedure Read_Array   (Array_Length : In Integer;
                        Out_Array    : Out Array_Int) Is Separate;
```

وبالتالي، يصبح العمل ضمن الزمن الحقيقي.

تعريف عمليات، على أنواع المعطيات المجردة(Abstract data type) :

كما نوهنا سابقاً في القسم السادس، يتصف النوع بمجموعة قيم، ومجموعة عمليات، يمكن تطبيقها على أغراض من هذا النوع. والأنواع الأولية (مثل الأعداد الصحيحة)، لها العمليات الخاصة بها. ولكن، يستطيع المبرمج أن يخلق أنواعاً خاصة به، مع العمليات المحددة القابلة للتطبيق على أغراض من هذا النوع. وسوف نناقش هذا الموضوع بالتفصيل، في القسم الحادي عشر. ومع ذلك، رأينا في مسألتي التصميم الأوليتين، كيف يمكن استعمال البرامج الجزئية لتنفيذ هذا التعريف. وتستخدم الحزم البرمجية لتعليب وزيادة ترابط التجريدات المنطقية، كما يلي:

```
Package Linked_list is
  Type List is limited private;
  Procedure Add    ( To   : in out List; Data : in Integer);
  Procedure Remove( From : in out List; Data : out Integer);
  Funcion Is_Null   ( The_List : in List) return Boolean;
      Null_list : exception;
    Private
      Type List is ...
End Linked_list;
```

وكما نشاهد، إننا استخدمنا الاجرائيات كأدوات بناء (Constructors) العمليات التي تطبق على الغرض والتوابع كأدوات اختيـار (selectors) العمليـات التـي ترجـع قيمة للغرض. وفي هذا المثال، نلاحظ أن توصيف الحزمـة البرمجيـة، يحتـوي أجـزاء التوصيف من البرامج الجزئية. ويجب إضافة الأجسام في جسـم الحزمـة. ففـي المثـال السـابق، اسـتخدمنا نـوع معطيـات مجـرد Liked_Lists . لائحـة يطبـق عليهـا ثـلاث عمليات واختبار، إذا كانت فارغة. والتفاصيل التنفيذية لهذه البرامج، تبقى مخفية في هذا المستوى.





التعابير والتعليمات
Expressions and Statements

الأسماء

القيم

التعابير

التعليمات



في بعض الأحيان، يجب تنفيذ خوارزميات عالية المستوى، ونطمح لأن يكون هذا التنفيذ واضحاً ومقروءاً. وكما أن أنواع المعطيات المجردة تتكون من أنواع أساسية، فإن البرامج الجزئية تتألف أيضاً من تعليمات بسيطة، تزودنا ببُنى من أجل التحكم بالخوارزميات وتنفيذها، وتسهيل حساب القيم. وفي هذا الفصل، سنفحص التعابير والتعليمات في لغة ADA.

٩ ـ ١ ـ الأسماء (Names) :

قبل تنفيذ أي عملية على أي غرض، يجب أن نكون قادرين على الرجوع إليه أو لمركباته بواسطة إسم. وبلغة ADA، فإن الإسم يعين بشكل صوري كياناً مصرحاً عنه، مثل غرض، أو رقم، أو نوع، أو نوع جزئي، أو برنامج جزئي، أو حزمة برمجية، أو مهمة، أو مدخل لمهمة، أو استثناء. وعلى سبيل المثال، ففي التصريحات التالية، إن كل معرّف يبدأ بحرف كبير، يعتبر إسماً مقبولاً في ADA:

```
type Process_Type is (Running, Ready, Blocked, Dead);
type Count is array (Process_Type) of Natural;
type Count_Name is access Count;
Process_State : Process_Type;
Scheduler_Table : Count;
Local_Schedule : Count_Name;
subtype Coefficient is Float digits 7 range -1.0..1.0;
subtype Size is Integer range 1..4;
type Matrix is array(Size,Size) of Coefficient;
Matrix_1,Matrix_2 : Matrix;
type Value is record
  Name : String(1..10);
  Location : String(1..10);
  Open : Boolean;
  Flow_Rate :Float;
end record;
subtype Total_Values is Integer range 1..100;
```

```
type Values is array(Total_values) of value;
Value_Index : Total_Values;
Value_Record:Values;
Pi : Constant :=3.141_592_65;
Is_Empty,Is_Active : Boolean;
Voltage_1,Voltage_2 : Float;
Count_1,Count_2: Total_Values;
type Radians is new Float;
functions Cos(Angle: in Radians) return Float;
```

وعندما نريد الرجوع إلى الكيان بأكمله، نستخدم، ببساطةٍ، إسمه (مثلاً Matrix_1 و Value_Index). بينما من أجل الرجوع إلى جزء من كيان مركب، يجب استخدام رموزٍ مختلفة، نعالجها في حينها. أما الآن، فسنرى أمثلةً بسيطةً عن ذلك:

```
Matrix_1(1,4)
Scheduler(Process_Type'Succ(Process_State))
Value_Record(37)
```

ففي المثال الأول، نحصل على العنصر الرابع من السطر الأول من الغرض Matrix_1 .

وفي المثال الثاني، وفي حال الكائن Process_State مختلف عن Dead، في هذه الحالة، سوف يتم تعين Process_Type'Succ(Process_State) ، لنحصل على Scheduler عند هذه القيمة.

أما في حال كون Process_State مساوياً لـ Dead، فسيكون هناك خطأ أثناء التنفيذ، وهو Constraint_Error، سببه عدم وجود لاحق لـ Dead (Succ(Dead) غير موجود).

وفي المثال الثالث، يتم الحصول على كافة مكوّنات التسجيلة ذات الرقم ٣٧ من Value_Record .

وكما يمكننا الرجوع لكيان واحد، يمكننا تسمية سلسلة متتالية من المكونات، في شعاع وحيد البعد يدعى بـ Slice. والأمثلة التالية مختارة من أسماء لـ Slices:

```
Scheduler_Table(Ready..Dead)        -- 3 Components
Value_Record(1..50)                 -- 50 Components
```

وتعتبر الـ Slices مفيدةً بشكلٍ خاص، في حال نقل كتل ضخمة من المعطيات، من شعاع لآخر. فعلى سبيل المثال، لنفترض التعليمة التالية:

```
Value_Record(1..20):=Value_Record(21..40);
```

فوفق هذه التعليمة، يتم نسخ الـ ٢٠ عنصر (ابتداءً من العنصر رقم ٢١، وحتى العنصر رقم ٤٠) من الشعاع Value_Record إلى الشعاع نفسه، في الأماكن ١ وحتى ٢٠، وبنفس الترتيب الذي كانت عليه.

ويمكن أن نحصل على تراكب في الـ Slices، كما في المثال التالي:

```
Value_Record(1..10) := Value_Record(6..15);
```

وفي هذه الحالة، يجب تعيين قيم Value_Record(6..15) أولاً، ومن ثمّ تنفيذ هذه التعليمة.

٩ ـ ٢ ـ القيم (Values):

يوجد في ADA نوعان من القيم، القيم السلّمية، والقيم المركبة.

من القيم السلّمية مايلي:

```
1_024                -- an integer numeric literal
0.398_829_138        -- a real numeric literal
Blocked              -- an enumeration literal
"Warehouse"          -- a character string
null                 -- null access value, referring to no object at all
'b'                  -- a character literal
16#FFE#              -- a base 16 number
```

وبالرجوع إلى الأسماء قليلاً، لاحظ كيف تمّ التصريح عن Scheduler_Table. ففي هذه الحالة، يمكن توليد قيم للمصفوفة، وفق تسلسل تنضيد المكونات، ووفق إحدى الطرق التالية:

```
(7, 3, 1, 0)  -- Positional notation
(Running =>7, -- equivalent named notation
 Ready => 3,
 Blocked => 1,
 Dead =>0)
(Running .. Dead =>0)  --- using a range
(Running | Dead =>0,    --- using a choice
 Ready   | Blocked => 1)
(Running | Ready..Blocked=>1,-- Combination of range and choice
 Dead =>3)
```

ووفق هذا، لاحظ صحة الخيارات المختلفة لإعطاء قيم للمصفوفة، ولكن من الأفضل دائماً، استخدام الترميز المسمى (named notation) ليكون البرنامج مقروءاً بشكل أفضل.

٩ ـ ٣ ـ التعابير (Expressions):

تستخدم التعابير من أجل حساب قيم جديدةٍ للأغراض، بالإعتماد على قيم أغراض محددة. ومن أجل هذا، يجب استخدام التعابير بشكلها الصحيح، لتجنب الوقوع بأخطاء.

وباستخدام العمليات الحسابية والمنطقية، وبعض الأسماء، يمكن تشكيل عدة تعابير. والأمثلة التالية، توضح ذلك:

```
Pi                  -- a simple expression
(b**2)-(4.0*a*c)    -- a more complex expression
char in 'A'..'Z'    -- a boolean expression
(2.789**4)+36.0     -- a static expression
```

وأي تعبير، له قيمة ونوع. وللحصول على تعبير بتطبيق عدة عمليات على عدة أغراض، يجب أن تكون الأغراض من نفس النوع. إذ يمكن تحويل نوع غرض لنوع آخر، من أجل تنفيذ عملية معينة عند الحاجة.

مثال:

ليكن لدينا المثال التالي:

```
WITH Text_IO; USE Text_IO;
with text_io;use text_io;
PROCEDURE orange_apple IS
type  orange is range 1..150;
type  apple is range 1..150;
orange_quantity:orange;
apple_quantity:apple;
apple_price,Orange_price:integer;
total_price : integer;
package int_io  is new  integer_io(integer);
use int_io;
package orange_io is new integer_io(orange);
use orange_io;
package apple_io is new integer_io(Apple);
use apple_io;
BEGIN
put("Enter the orange's quantity: ");
get(orange_quantity);
put("Enter the apple's quantity: ");
get(apple_quantity);
put("Enter The Orange's price: ");
get(Orange_Price);
put("Enter The Apple's price: ");
get(Apple_Price);
put("the apple quantity is: ");
put(apple_quantity);
new_line;
put("the orange quantity is: ");
put(orange_quantity);
```

```
new_line;
total_price:= Orange_price*integer(orange_quantity)
        +apple_price*integer(apple_quantity);
put("the total price is: ");put(total_price);new_line;
END orange_apple;
```

ووفق هذا المثال، لاحظ أنّه تمّ تعريف النوع Orange والنوع Appele، بأنّ كل منهما محدد بالمجال [1,150]. وبعد ذلك، تمّ تعريف عدة أغراض من مختلف الأنواع. وأردنا حساب قيمة الغرض Total_Price، بالإعتماد على معرفة قيم الاغراض Oarnge_quantity و Apple_Quantity وOrange_Price وApple_Price فلم نستطع، حتى أجرينا تغيراً على نوع Orange_quantity وعلى نوع Apple_Quantity، وذلك باستخدام التعليمة (Integer(Orange_quantity والتعليمة (Integer(Appele_Qauntity (أي تمّ تغير نوع كلّ من Orange_Quantity, Appele_quantity إلى النوع Integer قبل إجراء الحساب).

ويجب تجنب القسمة على 0، وإلا سيكون هنالك «خطأ حسابي»، وهو Numeric_Error. وبالنسبة للمصفوفات، يجب عدم تجاوز حدود المصفوفات، وكذلك، بالنسبة للأنواع المرقمة، وإلا سيكون هنالك الخطأ Constraint_Error.

وكما ذكرنا سابقاً، توجد في لغة ADA عدة معاملات، يمكن استخدامها في تعابير لغة ADA، إذ أنّ لغة ADA تعرّف ستة صفوفٍ من العوامل التي يمكن أن تؤثر بالمعاملات. وهذه الصفوف ما يلي:

```
        not     **     abs                   -- Heighest Precedence Operator
/       mod     *      rem                   -- Multiplying Operator
-               +                            -- Unary adding Operator
-       &       +                            -- Binary adding Operator
/=      <       =      <=      >      >=     -- Relational operator
or      xor     and                          -- Logical Operator
```

ويمكن إجراء تحميل زائد على أيٍّ مـن رمـوز هـذه العوامـل، مـاعدا الرمـز "=/" وذلك مثلما لاحظنا سابقاً.

فإذا كان لدينا تعبير له الشكل التالي: L Op R حيث أنّ:

– L غرض يساري من نوع ما (أو لا شيء).

– OP أحد العمليات المحددة في الجدول السابق.

– R غرض يميني من نوع ما.

فما نوع ناتج العملية؟.

إن الجدول التالي يجيب على هذا السؤال:

Operator	Operation	Operand Type		Result Type
**	Exponentiation	L: integer	R: integer>=0	Integer
		L: floating	R: integer	Floating
*	Multiplication	integer		Integer
		floating		Floating
		L: fixed	R: integer	Fixed
		L: integer	R: fixed	Integer
		L: fixed	R: fixed	Universal_fixed
/	Division	integer		Integer
		floating		Floating
		L: fixed	R: integer	Fixed
		L: fixed	R: fixed	Universal_fixed
mod	modulus	integer		Integer
rem	remainder	integer		Integer
+	unary identity	Numeric type		Numeric type
-	unary negation	Numeric type		Numeric type
abs	absolute values	Numeric type		Numeric type
not	unary logical	boolean		Boolean
	negation	array of		same array
		boolean		Type
+	addition	numeric type		numeric type
-	subtraction	numeric type		numeric type
&	catenation	one-dimensional		same array
		array types		Typ

		array and		same array
		component		Type
		component and		same array
		array		Type
		component and		any array type
		component		
=	equality	any type		Boolean
/=	inequality	any type		Boolean
<	less than	any scalar type		Boolean
		discrete array type		Boolean
<=	less than	any scalar type		Boolean
	or equal	discrete array type		Boolean
>	greater than	any scalar type		Boolean
		discrete array type		Boolean
>=	greater than	any scalar type		Boolean
	or equal	discrete array type		Boolean
in	membership	L: any scalar	R.Range	Boolean
		L: any scalar	L: Subtype	Boolean
			indication	
not in	nonmembership	L: any scalar	R:Range	Boolean
		L: any scalar	L: Subtype	Boolean
			indication	
and	Conjunction	boolean		Boolean
		array of		same array
		boolean		Type
and then	Conjunction	boolean		Boolean
	(short circuit)	array of		same array
		boolean		Type
or	inclusive	boolean		Boolean
	disjunction	array of		same array
		boolean		Type
or else	inclusive	boolean		Boolean
	disjunction	array of		same array
		boolean		Type
	(short circuit)			
nor	exclusive	boolean		Boolean
	disjunction	array of		same array
		boolean		Type

وإن تنفيذ العمليات (*,/,mod,rem) على الأعداد الصحيحة، تعطي نتائج دقيقة، بينما على الأعداد الحقيقية، تعطي نتائج تقريبية.

وإن العملية rem، تعيد باقي قسمة L على R، وإشارة الناتج من إشارة L، بينما mod، تعيد باقي قسمة L على R، وإشارة الناتج من إشارةR.

أمثلة:

L	R	L rem R	L mod R
12	5	2	2
14	5	4	4
12	-5	2	-3
14	-5	4	-1
-12	5	-2	3
-14	5	-4	1
-12	-5	-2	-2
-14	-5	-4	-4

٩ - ٤ - التعليمات (Statements):

تحتوي لغة ADA على مجموعة تعليمات، قادرةٍ على خلق الخوارزميات. وبما أنّ ADA تعتبر لغةً بنيوية، فإنها تتضمن التعليمات التالية:

التحكم التسلسلي (Sequential control):

تضم تعليمات التحكم التسلسلي مايلي:

- Assignment.
- Block.
- Null.
- Return.
- Procedure Call.
- Goto.

التحكم التكراري (Iterative controle):

وتضم تعليمات التحكم التكراري ما يلي:

- Basic Loop.
- for Loop.
- While Loop.
- Exit.

التحكم الشرطي (Conditional control) :

وتضم تعليمات التحكم الشرطي ما يلي:

- Case.
- If.

تعليمات مختلفة (Other statments):

وتضم التعليمات الأُخرى ما يلي:

- – Abort . – Accept . – Code .
- – Delay . – Entry Call .
- – Goto . – Raise . – Select .

التحكم التسلسلي (Sequential control):

في التحكم التسلسلي، يتم تنفيذ التعليمات الواحدة بعد الأُخرى، بطريقة خطية. وتحتوي ADA على التعليمات التسلسلية التالية:

الإسناد (Assignment):

إن الشكل العام للإسناد كما يلي: Obj := Expr;

حيث Obj غرض متغير، وExpr تعبير نوع، نتيجته من نوع الغرض Obj. وعند حساب قيمة التعبير Expr، يتم تغيير قيمة الغرض Obj، لتصبح هي قيمة التعبير Expr، بدلاً مما كانت عليه قبل تنفيذ هذه التعليمة.

```
Counter              := Counter+1;
Matrix_1             := Matrix_2;
Birth_Day.Year       :=1955;
Value_Record(Count_1).Open:=True;
Value_Record(1..10):= Values'(1..10 => (Name => "Spare  ",

Location   => "Warehouse ",
Open       => False,
Flow_Rate  => 0.0 ));
```

لاحظ أنه في المثال الأخير، قد تم الإسناد لمجموعة (الـ 10 عناصر الأولى) من العناصر، بتعليمة واحدة، لنفس القيمة.

الكتلة (Block):

تستخدم الكتلة البرمجية لتعليب مقطع من ترميز، من أجل التصريح عن أغراض محلية، أو إستثناءات محلية، أو أنواع، لا يمكن استخدامها من قبل وحدات برمجية أخرى.

والكتلة، لها الشكل التالي:

```
Name_Block:
 Declare
  Declarations_Part;
Begin
 Statements;
End Name_Block;
```

فمن هذا الشكل، نجد أنّ الكتلة تتألف من:

- إسم: يجب اختياره بحيث يكون مقبولاً.
- تعليمة Declare.
- «قسم التصريحات» الخاص بهذه الكتلة.
- Begin: للإشارة إلى «بدء» التعليمات الخاصة بالكتلة.
- «مجموعة تعليمات» تخص الكتلة.
- End: للإشارة إلى «انتهاء» التعليمات الخاصة بالكتلة.

وفي حال عدم وجود «تصريحات» يمكن الإستغناء عن تعليمة Declare .

ويمكن لكتلةٍ أن تحتوي كتلةً أو أكثر في داخلها.

ويمكن أن نلاحظ أن فكرة المحلية، والعمومية، واضحة بشكلٍ جيد في الكتـل. فتعريف متحول محلي في برنامج ما، يلغي تأثير متحول عام ضمن الكتلة المعرّف بها.

مثال:

```
With Text_IO; Use Text_IO;
Procedure Example_Block is
Package Int_IO Is New Integer_IO(Integer);Use Int_IO;
   x : integer := 5;
   y : integer := 8;
  Begin
    put("X="); put(x); New_Line;          --- X=5
    put("Y="); put(y); New_Line;          --- Y=8
    put("X rem Y="); put(x rem y); New_Line  --- X rem Y=5
    Swap:
      Declar
       y : integer := 39;
       z : integer;
      Begin
       z:=x;
       x:=y;
       y:=z;
       put("X="); put(x); New_Line;  ---  X=39
       put("Y=");put(y); New_Line;  --- Y=5
       put("X rem Y="); put(x rem y); New_Line; -- X rem Y=4
      End Swap;
     put("X="); put(x); New_Line;    --- X=39
     put("Y="); put(y); New_Line;    ---  Y=8
     put("X rem Y="); put(x rem y); New_Line; --- X rem Y=7
   End Example_Block;
```

وهذا برنامج متكامل، تمّ من خلاله تبيان توضع الكتلة، وكيفية استخدامها، والمتغيرات العامة، والمحلية.

لاحظ أنّ:

- x متحول عام داخل وخارج الكتلة، وكيف تمّ تغيير قيمته في داخلها.
- y متحول عام قبل بدء الكتلة وبعدها، ويأخذ القيمة 8. أما داخل الكتلة فيعتبر متحولاً محلياً قيمته 39، ويتم تبديلها مع قيمة المتحول العام x لتصبح 5 .
- x rem y يمثل ناتج قسمة x على y .
- لاشيء (Null): عندما تصادف هذه التعليمة بالبرنامج، يعني عدم تنفيذ شيء.

الغاية من التعليمة Null :

- تسمح بمرور مؤشر التحكم للتعليمة التي تليها بسهولة.
- تُصبح قراءة البرنامج أكثر وضوحاً.

وتستخدم هذه التعليمة في حالة الـ:

- Case من أجل اختيارات غير مهمة.
- Record عند التصريح، للإشارة إلى قسم فارغ من متغير.
- في جسم برنامج جزئي Stub.

الرجوع (Return):

بشكل أساسي، تنهي تعليمة Return تنفيذ برنامج جزئي. وبالإضافة لذلك، تعيد قيمة البرنامج الجزئي الموجودة ضمنه، إذا كان تابعاً فرعياً.

مثال:

```
function issued(Value : in integer) return Boolean is
begin
 if (Value rem 2)=0 then
   return False;
 else
```

```
   return True;
 end if;
end issued;
```

فوفق هذه التابع الفرعي، يتم فحص قيمة الدخل Value، فإذا كانت فردية، فإنه ستعاد قيمة التابع بـ False، وإلاً ستعاد قيمة التابع بـ True .

فإذا كان البرنامج الجزئي يمثل تابعاً فرعياً، عنده يجب أن يلي تعليمة Return تعبير ما، نوع نتيجته من نوع إسم التابع الفرعي، وبالتالي، عند الرجوع بتعليمة Return، يتم حساب قيمة التعبير، وإسناده لقيمة التابع.

مثال:

```
function Is_Odd(Value : in integer) return Boolean is
begin
  return (Value rem 2) /= 0;
end Is_Odd;
```

لهذا البرنامج الفرعي، نفس عمل البرنامج الفرعي السابق، ولكن بصيغة أخرى.

استدعاء برنامج جزئي (Subprogram Call):

لقد تم شرحها مفصلاً في فصل البرامج الجزئية.

إذهب إلى (Goto):

لا يمكن اعتبار هذه التعليمة، بشكل كامل، تعليمة تحكم تسلسلية، لكن تصنيفها ضمن هذه المجموعة أفضل من تصنيفها ضمن بقية المجموعات.

إنّ تعليمة Goto، ليست أساسية بلغة ADA لتنفيذ خوارزمية معينة، إذ يمكن أن يستعاض عنها بتعليمات أُخرى لتنفيذ الخوارزمية المعينة، ومن المفضل الحد من استخدام هذه التعليمة.

وتستخدم هذه التعليمة كما يلي: **goto label**

حيث label، رمزٌ لسطر تعليمة ضمن البرنامج، إذ أنّ label محاطة بـ >> و<<.

مثال:

```
WITH Text_IO; USE Text_IO;
PROCEDURE Goto_Example IS
package int_io is new integer_io(integer);
use int_io;
y,x: integer;
BEGIN
put_line("Enter Two Integer Number.");
get(x);get(y);
<<another_one>>  put_Line("----------------");
put("X=");put(x);new_line;
put("Y=");put(y);new_line;
if y/=0 then
 goto calc;
else
 put_line("Renter Y");
 get(y);
 goto another_one;
end if;
<<calc>>  put("X rem Y=");put(x rem y);new_line;
END Goto_Example;
```

هذا مثال متكامل، يستخدم تعليمة goto مرتين في مكانين مختلفين، وذلك حسب قيمة المتغير y. فحاول أن تكتب البرنامج، وتفحصه على مترجم لغة ADA.

التحكم الشرطي (Conditional control):

يستخدم التحكم الشرطي، لاختيار تعليمة واحدة من عدة تعليمات. وهذا الإختيار، يعتمد على بعض الشروط، أو التعابير.

تعليمات التحكم الشرطي تضم If وCase.

: If

وفق تعليمة If يتم اختيار تعليمة (أو ولا تعليمة) من بين عدة تعليمات من أجل تنفيذها باعتماد قيمة منطقية لشرط أو أكثر. وتمثل هذه الشروط تعابير منطقية، والتي لها القيمة True أو القيمة False. ويوجد ثلاثة أشكال أساسية لتعليمة If الشرطية. وفيمايلي هذه الأشكال:

- **الشكل البسيط:** ومثال ذلك مايلي:

```
if Count_1 < 5 then       --  a simple if-then
  Count_1:=9;
end if;
```

- **الشكل If-Then-Else :** ومثال ذلك مايلي:

```
if Value_Record(1).Open then -- an If-Then_Else Construct
  Value_Record(2).Open:=True;
  Value_Record(3).Open:=False;
  else
    Value_Record(2).Open:=False;
    Value_Record(3).Open:=True;
end if;
```

- **بنية If المتوازية:**

```
if Voltage_1>Voltage_2 then  --  A Parallel if structure
   Voltage_1:=Voltage_2;
  elsif Voltage_1<Voltage_2 then
   Voltage_2:=Voltage_1;
enif;
```

لاحظ وفق الشكل الأخير، أنّه لم يتم فحص حالة Voltage_1=Voltage_2.

ولاحظ أنّه وفق الأشكال الثلاثة، لابد من ;end if لإنهاء تعليمة if ، لكن من أجل elsif يجب عدم وضع end if.

ففي الحالة الأولى، تتم مقارنة الغرض Count_1 مع القيمة 5. فإذا كانت قيمته أصغر من 5، عندئذ سيتم تغيرها للقيمة 9، وإلاَّ، تبقى على ما كانت عليه.

وفي الحالة الثانية، يتم فحص قيمة Value_Record(1).Open. فإذا كانت مساوية لـ True، عندها ستنفذ التعليمتين التاليتين:

```
Value_Record(2).Open:=True;
Value_Record(3).Open:=False;
```

أما إذا كانت قيمة Value_Record(1).Open مساوية لـ False، عندها ستنفذ التعليمتين التاليتين:

```
Value_Record(2).Open:=False;
Value_Record(3).Open:=True;
```

وفي الحالة الثالثة تُقارن قيمة الغرض Voltage_1 مع قيمة الغرض Voltage_2، فإذا كانت قيمة الغرضVoltag e_1 أكبر تماماً من قيمة الغرض Voltage_2، عندها ستنفذ التعليمة Voltage_1:=Voltage2. أما إذا كانت قيمة الغرض Voltage_2 أكبر تماماً من قيمة الغرض Voltage_1، عندها ستنفذ التعليمة Voltage_2:=Voltage_1 .

Case :

إن بنية التعليمة Case تشبه بنية التعليمة If ، ولكن لا تشبه التعليمة If، إذ أنّه بواسطة التعليمة Case، يمكن اختيار تعليمة من عدة تعليمات، وذلك، بالإعتماد على قيمة تعبيرٍ متقطع (والتعبير المتقطع، هو كل تعبير يعيد قيمةً من النوع الصحيح، أو النوع المرقم).

مثال:

فيمايلي برنامج متكامل يقرأ عددين صحيحين، ومحرف يمثل إشارة. وحسب المحرف المقروء، تُنفذ العملية الحسابية الموافقة على العددين، أو لا تنفذ في حال كون المحرف المقروء لا يرمز لعملية حسابية:

```
WITH Text_IO; USE Text_IO;
PROCEDURE arit_op IS
```

```
package int_io is new integer_io(integer);use int_io;
operation:character;
Error:boolean;
first,second_result:integer;

procedure Arith_Operation(first : in integer;
                    second_result: in out integer;
                    operation:character;Error:out boolean) is
 Begin
  Error:=true;
  case operation is
   when '+' => second_result := first + second_result;
   when '-'  => second_result := first  - second_result;
   when '*' =>  second_result := first * second_result;
   when '/' | '%' =>          -- /  result of division , %  rest of division
    if second_result=0 then
     put_line("Division By Zero !!");
     Error := false;
    else
     if operation='/' then    -- result of division
      second_result := first  /  second_result;
      else                        -- rest of division
       second_result := first rem second_result;
     end if;
    end if;
   when others=> put_line(" Illigal Operation ! ");
                  Error := false;
  end case;
 End Arith_Operation;
BEGIN        -- arit_op
 loop
  put("Enter First_Integer_Number Operation ");
```

```
 put_Line(" Second_Integer_Number");
 get(first);
 get(operation);
 get(second_result);
 exit when (first=second_result ) and (first=0);
put(integer'image(first) & operation &
    integer'image(second_result) & "=");
 arith_operation(first,second_result,operation,Error);
 if Error then
  put_line(integer'image(second_result));
 end if;
end loop;
END arit_op;
```

من أجل هذا، تمّت كتابة إجرائية، دخلها ما يلي:

- العددان الصحيحان first وsecond_result، الممثلان للعددين الذين ستنفذ عليهما العملية، إن أمكن ذلك.
- الرمز المحرفي Operation، يرمز للعملية الحسابية.

أما خرجها، فهو ما يلي:

- الغرض المنطقي Error، الذي يشير إلى إمكانية تنفيذ العملية الحسابية على العددين الصحيحين، أم لا .
- العدد الصحيح Second_result، والذي يمثل ناتج العملية إذا كانت Error=True، أو يبقى كما كانت قيمته قبل استدعاء الإجرائية، إذا كانت Error=False .

لاحظ هنا، أنّه تمّ استخدام التعليمة If-Then-else ضمن تعليمة Case .

وبالتالي، إن الشكل العام لتعليمة Case هو ما يلي:

```
Case Expression is
 When Value1 => Statements1
 When Value2 => Statements2
```

```
 ...
 When ValueN => StatementsN
 When Others => Statements_Others
End Case;
```

حيث Expression هو تعبير متقطع، يتم حساب قيمته ومقارنتها مع القيم Value1, Value2, ... بالترتيب إعتبارا من Value1، حتى تتطابق قيمة التعبير مع القيمة المحددة، عندها يتم تنفيذ مجموعة التعليمات الموافقة للقيمة. فمثلاً، إذا كانت قيمة Expression متطابقة مع Value4 عندها ستنفذ مجموعة التعليمات Statements4 فقط، ومن ثم الخروج من Case.

وفي حال عدم تطابق قيمة Expression مع أيّ من القيم Value1, Value2, ..., ValueN، عندها سيتم تنفيذ مجموعة التعليمات Statements_Others، وذلك في حال طلب When Others => Statements_Others، إذ أنّه من الممكن الإستغناء عن هذه الأخيرة.

ويمكن التنويه هنا، بأنّ تعليمة Case تستخدم في حال وجود شروط تعتمد على قيم سلّمية، بينما تعليمة If يجب أن تستخدم في الحالات الأخرى، بما في ذلك التعابير المنطقية المعقدة.

التحكم التكراري (Iterative control) :

يمكن تعريف التحكم التكراري، بأنّه التحكم الذي ينفذ مجموعةً من التعليمات أكثر من مرة، أو لا ينفذها، وذلك وفق شروطٍ معينة، تتحكم بعدد مرات التكرار.

وإن تعليمات التحكم التكرارية، تضم ما يلي:

- Basic Loop .
- for Loop .
- While Loop .
- Exit .

وفيما يلي، شرح مفصل لكل من هذه التعليمات:

وقبل كل شيء، يجب شرح تعليمة Exit، والتي تفيد في الخروج من الحلقة، عند تحقيق شرط معين. وتستخدم هذه التعليمة، وفق أحد الأشكال التالية:

- Exit : فوفق هذا الشكل، يتم الخروج من الحلقة الحالية.
- Exit Outer : ووفق هذا الشكل، يتم الخروج من الحلقة المسماة Outer
- Exit When Condition : ووفق هذا الشكل، يتم الخروج من الحلقة الحالية، عند تحقيق «الشرط» Condition.
- Exit Outer When Condition : ووفق هذا الشكل، يتم الخروج من الحلقة المسماة Outer، عند تحقيق «الشرط» Condition.

وفي تعليمات التحكم التكراري، سنستخدم تعليمة Exit أكثر من مرة، وسنُظهر دورها بشكل جيد.

Basic Loop:

تعتبر هذه التعليمة من أبسط تعليمات التحكم التكراري، ويمكن أن تكون لا نهائية.

وهذه التعليمة، لها الشكل التالي:

```
Loop
  -- Sequence of statements
end Loop;
```

Sequencc of statements تمثل «مجموعة التعليمات» التي ستكرر ضمن الحلقة.

مثال:

فيما يلي، سنكتب تابعاً فرعياً تحسب العاملي لعدد صحيح موجب، باستخدام Basic Loop، كما يلي:

```
Function Factorial(A_Number : in Natural) return Natural is
 Numb1, Numb2 : Natural;
Begin
```

```
 Numb1 := A_Number;
 Numb2 := A_Number;
Loop
  Exit When Numb1<2;
  Numb1:=Numb1-1;
  Numb2:=Numb2*Numb1;
End Loop;
return Numb2;
End Factorial;
```

سيتم تنفيذ هذه الحلقة، حتى يتحقق الشرط Numb1<2، عندها تمثل قيمة Numb2، قيمة عاملي العدد A_Number، وينتهي تنفيذ الحلقة.

ويمكن تداخل الحلقات التكرارية مع بعضها، ويمكن تسميتها بأسماء مختلفة، كما يلي:

```
Outer_Loop:
```

ويمكن الخروج من «الحلقة الداخلية» Inner_Loop إلى «الحلقة الخارجية» Outer_Loop بالتعليمة Exit، وفق أحد أشكالها الأربعة الآنفة الذكر، ضمن المنطقة B. وكذلك الأمر، يمكن الخروج من «الحلقة الخارجية» Outer_Loop، إلى خارج «الحلقة الخارجية» Outer_Loop، من المنطقة A، والمنطقة C، وفق أحد أشكال التعليمة Exit الأربعة.

أما إذا كنا في المنطقة B، وأردنا الخروج إلى ما بعد «الحلقة الخارجية» Outer_Loop، عندها يجب استخدام تعليمة Exit المسمّاة أو الشرطية المسماة.

: for Loop

إن «حلقة من أجل» For Loop، لها الشكل التالي:

```
For Variable in First_Limit..Last_Limit
 Loop
  --Sequence of statements
 end Loop;
```

حيث Variable، يمثل إسماً متغيراً، ليس بالضرورة مصرح عنه من قبل، و First_Limit وLast_Limit، يمثلا حدود التكرار، وبالتالي، ستتكرر الحلقة عدداً من

المرات، يتعين بـ First_Limit وlast_Limit. وفي هذه الحالة، يجب أن يكون First_Limit أصغر من Last_Limit، ويجب أن يكونا من نوع متقطع. وإن القيمة الأولى لـ Variable تبدأ بـ First_Limit، والقيمة النهائية تنتهي بـ Last_Limit.

ويمكن أن يبدأ المتغير Variable بـ Last_Limit، وينتهي بـ First_Limit إذا أضفنا كلمة Reverse، كما يلي : For Variable in Reverse First_Limit..Last_Limit

ويمكن الخروج من «الحلقة من أجل» For Loop أيضاً، باستخدام أحد أشكال Exit، وذلك حسب الضرورة.

مثال : سنعيد كتابة التابع الفرعي Factorial، باستخدام «حلقة من أجل» For Loop، كما يلي:

```
Function Factorial(A_Number : in Natural) return Natural is
 Numb : Natural;
Begin
 Numb := 1;
For Index in 2 .. A_Number
Loop
  Numb:=Numb*Index
End Loop;
return Numb;
End Factorial;
```

ويمكن الإستعاضة عن First_Limit وLast_Limit بتعابير تضم مجالاً من القيم المتقطعة، ومحددة.

مثال ذلك Values'range، الذي يشير إلى مجال النوع Values، Total_Values، الذي يمثل مجالاً من الأعداد الصحيحة...

:While Loop

إن تعليمة While Loop، لها الشكل التالي:

```
While Condition
Loop
```

```
 Sequence Of Statements
End Loop;
```

حيث Condition يمثل «شرط» الخروج من الحلقة التكرارية While Loop. ويتم الخروج من هذه الحلقة، عندما تكون قيمة «الشرط» Condition مساويةً لـ False. وأيضاً، يمكن الخروج منها باستخدام التعليمة Exit، وفق أحد أشكالها الأربعة، وحسب الحاجة.

مثال:

سنعيد كتابة التابع الفرعي مرةً ثالثة باستخدام While Loop، ليصبح على الشكل التالي:

```
Function Factorial(A_Number : in Natural) return Natural is
    Numb1, Numb2 : Na;
Begin
    Numb1 := A_Number;
    Numb2 := A_Number;
While Numb1>1
Loop
     Numb1:=Numb1-1;
     Numb2:=Numb2*Numb1;
End Loop;
return Numb2;
End Factorial;
```

ويمكن في وحدة برمجية ما، وكما في جميع لغات البرمجة، تداخل الحلقات مع بعضها البعض، وذلك حسب الحاجة.



10

مسألة التصميم الثانية
متابعة

عودة إلى المسألة

تقييم الأغراض

زرع كل غرض



الآن، وبعد دراستنا للبرامج الجزئية والتعليمات بـ ADA، والتي تعتبر كوسائل للتعبير عن العمل والتحكم، يمكننا إكمال حل المسألة الذي بدأناه في الفصل٧.

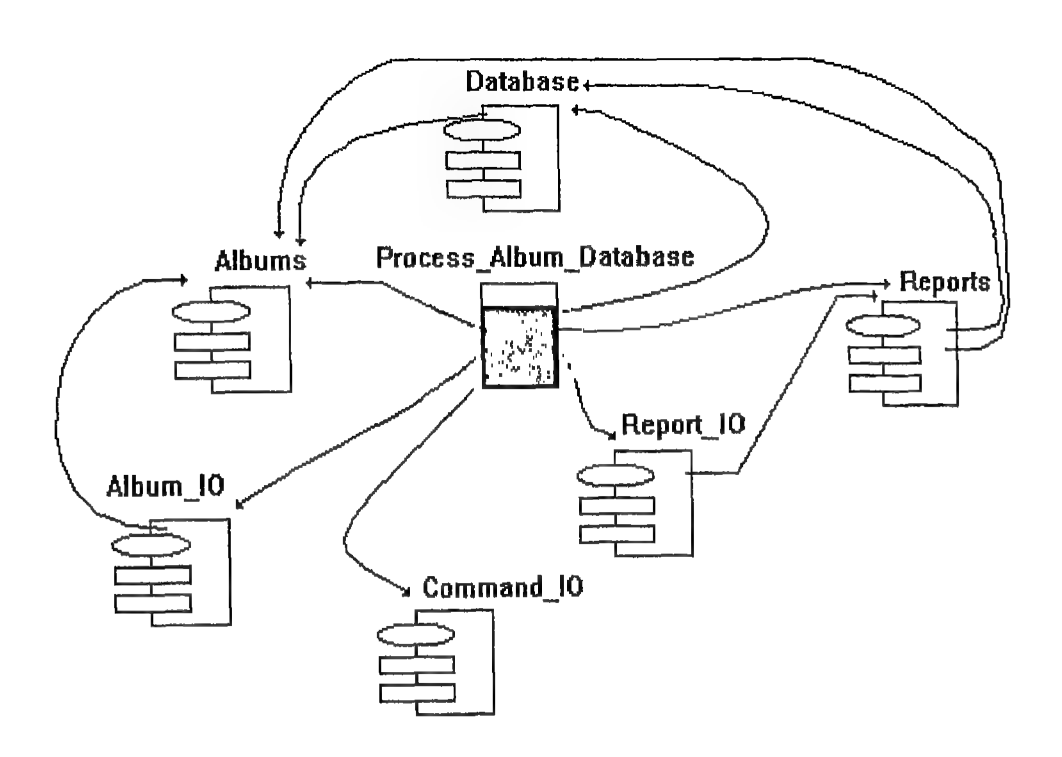
١٠ ـ ١ ـ عودة إلى المسألة (The Problem Revisited) :

دعنا نراجع تعريف المسألة لقاعدة المعطيات التي قدمناها سابقاً. وبشكل محدد، تتمثل مهمتنا ببناء نظام قاعدة معطيات، يحتفظ بتسجيلات حول كل ألبوم في مجموعتنا. ومن أجل كل ألبوم، نحتفظ بمعلومات حول العنوان، والفنان، وأسلوب الموسيقا، وسنة الإصدار،

وإسم وطول الأغنية في الألبوم. ونحتاج لأن نكون قادرين على خلق، وفتح، وإغلاق قاعدة المعطيات. ونفس الشيء، بالنسبة لإضافة، وحذف، وتغيير مداخل الألبوم الشخصي. وأخيراً، نريد أن نكون قادرين على صنع تقارير حول الألبومات، التي تمّ وصفها بقاعدة المعطيات الخاصة. ومن أجل هذه التقارير، نحتاج لأن نكون قادرين على اختيار تسجيلات، مؤسسة حسب بعض المعايير. فعلى سبيل المثال، نرغب بإيجاد جميع الألبومات، التي تمّ إصدارها في سنةٍ خاصة. وأيضاً، نحتاج لأن نكون قادرين على فرز التسجيلات.

والشكل ١٠-١، يوضح بنية حلّنا. والحزمة البرمجية Albums، تمثل التجريد المركزي؛ وتوفر هذه الوحدة عدة أنواع غير معلبة، توصف المعلومات التي ذكرناها من أجل كل ألبوم. وتمثل Database آلةٍ حالة-مجردة، التي تلعب دور مخزنٍ لجميع تسجيلات الألبوم. وتصدر هذه الوحدة عمليات مثل Open, Close, Delete كمكرّرات، وكذلك، مكرّر يسمح لنا بتصفح كل التسجيلات في قاعدة المعطيات، دون تخريب حالته.

ويمثل Reports، آخر وحدة أساسية في هذا المستوى من التجريد. فإنّه يعتمد على Albums و Database، ويوفر نوع معطيات مجرد، يُدعى Report . وتُصدر هذه الحزمة البرمجية عدة إجرائيات محملةً زائداً تُدعى Find، والإجرائية Sort ومكرر.



الشكل ١٠ ـ ١. تصميم Process_Album_Database

١٠ ـ ٢ ـ تقييم الأغراض (Evaluate the Objects) :

نقدم في هذا المقطع ، ثلاث موجّهات لتقييم الأغراض. وغالباً ، ما يكون تحديد وتعريف الأغراض صعباً. وهذه الخطوة الإضافية من تقييم الأغراض ، تساعدنا على ربح الوثوقية التي أولاً ، حددنا بها أغراض مسألتنا ، وثانياً ، عرّفنا بها دقة الأغراض. وتُلخص الموجّهات الثلاث ، كما يلي :

- *تخفيض الإرتباط.* إذا اعتمد غرض على أكثر منه إلى ٧ أغراض أخرى ، من المحتمل أن يوجد أغراض أكثر ، يجب تحديدها.
- *زيادة التماسك.* يجب أن يمثّل كل غرضٍ مستوى واحداً من التجريد فقط.

• *فصل البِّناءات والمختارات*. ولزيادة قابلية الفهم، يجب أن تكون العمليات التي تغير قيم غرض، والعمليات التي ترى قيم غرض، ببساطة واضحة .

تخفيض الإرتباط (Minimize Coupling) :

عند تأسيس الرؤية بين غرضين، نُنتج مخططاً كما هو الحال في الشكل ١٠-١. وهنا نرى عدم اعتماد غرض (ماعدا البرنامج الجزئي الرئيسي) على أكثر من غرضين. وهذا يعني، أنه ولا يوجد أكثر من سهمين يخرجان من أيّة حزمة برمجية. ويكون هذا «التفكك» (decoupling)، أو فصل الأغراض، حرجاً للصيانة.

وفي النظم الضخمة، حيث من الشائع لمئات من الأغراض المراد تحديدها، يمكننا الحصول على حالة، يكون فيها الغرض معتمداً على العديد من الأغراض الأخرى. وهذا يشير عادةً، بأنّ الغرض ضخم، ويحتاج لتفكيكٍ أكثر إلى أغراض أصغر.

زيادة التماسك (Maximize Cohesion) :

عند تأسيس واجهات تخاطب الأغراض، نبدأ بجعل تصريحاتنا ملموسة، وذلك بالتصريح عن الأنواع. وإذا تطلبت واجهة تخاطب غرض، التصريح لعديدٍ من الأنواع، فمرةً ثانية، يكون الغرض ضخماً جداً. وهذا صحيح خصوصاً، إذا صرحت واجهة التخاطب عن أنواعٍ خاصة، أو خاصة محدودة عديدة. والأغراض التي تصرح عن أنواع خاصة عديدة، غالباً ما تحاول أن تعمل في مستويين من التجريد، مرةً واحدة.

فلنعتبر الغرض Albums. وقد تمّ تعريف واجهة تخاطبه، كتوصيف حزمة برمجية. ويعرّف التوصيف تسعة أنواع، ونوعاً جزئياً واحداً. ويمكننا بالتأكيد، تعريف كل واحدٍ من هذه الأنواع، في غرض خاص. فعلى سبيل المثال، الغرض Years، يمكن تعريفه كما يلي:

```
package Years is
   type Year is range 1877..Integer'Last;
end Years;
```

لاحظ، على أية حال، بأنّ معظم الأنواع المعرفة في Albums، تكون في الواقع أساسية. وهذا يعني، أن لها تصريحات بسيطة. والتصريح عن العديد من الأغراض الصغيرة، مثل Years السابقة، سيعقّد تصميمنا كثيراً، دون ظهور أية فائدة. وبمعنى آخر، زيادة عدد الأغراض يعقّد التصميم، بصرف النظر عن بساطة الأغراض الشخصية.

وعند التقييم المتعاقب، مثل أن تكون Years غرض في تصميمنا، يجب أن نرجع إلى الأهداف الأساسية لهندسة البرمجيات، في الفصل ٢ . فهل سيكون تصميمنا قابلاً للتغيير كثيراً، إذا تمّ التصريح عن Years، وكيف يبدو أن Year ستتغير مؤخراً؟ بالإضافة لذلك، كيف يبدو أنّ Year، يمكن إعادة استخدامها في أيّ مكان آخر؟ ففي هذا المثال، قررنا بألا تتغير Year، وألا يعاد استخدامها. وهكذا، ستزيد قابلية الصيانة، بتصغير عدد الأغراض في تصميمنا.

فصل البنّاءات والمختارات (Separate Constructors and Selectors) :

يمثل البنّاء، عملية تغيير غرضاً. فعلى سبيل المثال، Database.Add تُغيّر Database؛ إذ أنّ «إضافة» Add، تُغيّر القيمة المجردة من قاعدة المعطيات. «أما» المختار، فلا يغير القيمة، ولكن في الواقع يرى واجهة القيمة. وإنّ Size وvalue_Of هما مختاران من الغرض Database. والعمليات التي تكون بنّاءة ومختارة، هي بشكل عام، معقّدة الفهم والإستخدام. وسنحاول تعريف عمليات، إما أن تكون بنّاءات أو مختارات، ولكن ليس الإثنين معاً. والمثال الأكثر شيوعاً لعملية بنّاءة ومختارة، يتمثل بعملية المكدّس Pop، عند تعريفه كما يلي:

```
Procedure Pop (Top_Item : out Item;
Off_Of   : in out Stack);
```

يغير Pop معاً المكدّس بحذف عنصره الذي يقع في القمة، ويرى المكدّس القديم، بإعادة العنصر الذي يقع على القمة.

وهناك طريقتان إضافيتان، ستتم مناقشتهما في الفصل ١٣ .

١٠ ـ ٣ ـ زرع كل غرض (Implement Each Object) :

إن ما تبقى علينا عمله، هو التنفيذ البرمجي للبرنامج الرئيسي، وأجسام كل الحزم البرمجية، التي تمّ ذكرها في الشكل ١٠ ـ ١.

دعنا نبدأ بجذر النظام، Process_Album_Database . وهذه الإجرائية دون معاملات، تُستخدم كبرنامج رئيسي. ومثلما قررنا في الفصل ٧، يتمثّل نشاط هذا البرنامج الجزئي، بتنسيق جميع ردود الأفعال مع المستخدم. وهكذا، سنكتب جسمه كحلقة أساسية. وداخل الحلقة، يمكننا استدعاء Command_IO.Get للحصول على طلب المستخدم. ثمّ، يمكننا تطبيق تعليمة Case، لاختيار سلسلة أفعال حسب ذلك الطلب. فعلى سبيل المثال، إذا طلب المستخدم البدء بتقرير جديد، سيتمثل فعلنا باستدعاء الإجرائية Reports.Initialize من أجل غرض تقرير محدد. وتتطلب بعض الأفعال، استجابةً طفيفةً من المستخدم. فعلى سبيل المثال، إذا أراد المستخدم فتح قاعدة معطيات، يجب أن نستخدم إجرائيةً من Text_IO لقراءة سطر دخل من المستخدم. وبشكلٍ مشابه، إذا أراد المستخدم اختيار عناصرٍ من تقرير، يجب في البدء، استدعاء Reports.Get للحصول على معيار البحث من المستخدم.

وجميع ردود الأفعال هذه مع المستخدم، تتطلب منا التصريح عن عدد من الأغراض محلياً. فعلى سبيل المثال، إذا بدأ المستثمر البحث حسب إسم الفنان، فإنه يلزم غرض محلي يحفظ قيمة إسم هذا الفنان. وبالفعل، يجب أن نملك غرضاً لكل صفٍ ممكن من دخل المستخدم.

وأخيراً، يشير تصميمنا المعبر عنه في الشكل ١٠ ـ ١، بأنّ هذا البرنامج الرئيسي، يستورد عدة وحدات. ومن أجل هذا السبب، يجب أن نملك عبارة سياقٍ طويلة نسبياً، لجعل الوحدات مرئية.

ويمكننا كتابة جسم هذه الإجرائية الرئيسية، كما يلي:

```
With Text_IO, Albums, Database, Reports, Album_IO, Report_IO,
Command_IO;
```

```
Procedure Process_Album_Database is
The_Command : Command_IO.Command;
The_Parameter : String(1..80);
Last_Character : Natural;
The_Album : Albums.Album;
The_New_Album : Albums.Album;
The_Title : Albums.Title ;
The_Artist : Albums.Artist ;
The_Style : Albums.Style ;
The_Year : Albums.Year ;
The_Number : Albums.Number ;
The_Song : Albums.Song ;
The_Length : Albums.Length ;
The_Category : Reports.Category;
The_Order : Reports.Order ;
The_Report: Reports.Report;
The_Iterator : Reports.Iterator ;
  Begin
 Loop
Command_IO.Get(The_Command);
case The Command is
 when Command_IO.Quit =>
        Text_IO.Put_Line("Leaving the database manager.");
        exit;
      when Command_IO.Create =>
        Text_IO.Put("Enter the database name: ");
        Text_IO.Get_Line(The_Parameter, Last_Character);
        Database.Create(The_Parameter(1..Last_Character));
```

```
        Text_IO.Put_Line("The Parameter(1..Last_Character)
                          & " has been created");
    when Command_IO.Open =>
        Text_IO.Put("Enter the database name: ");
        Text_IO.Get_Line(The_Parameter, Last_Character);
        Database.Open(The_Parameter(1..Last_Character));
        Text_IO.Put_Line("The Parameter(1..Last_Character)
                         & " is open");
    when Command_IO.Close =>
        Database.Close;
        Text_IO.Put_Line("Database has been closed");
    when Command_IO.Add =>
        Text_IO.Put_Line("Enter an album description: ");
        Album_IO.Get(The_Album);
        Database.Add(The_Album);
        Text_IO.Put_Line("Item added to the database");
    when Command_IO.Delete =>
        Text_IO.Put_Line("Enter an album description: ");
        Album_IO.Get(The_Album);
        Database.Delete(The_Album);
        Text_IO.Put_Line("Item removed from the d");
    when Command_IO.Modify =>
        Text_IO.Put_Line("Enter an album description: ");
        Album_IO.Get(The_Album);
        Text_IO.Put_Line("Enter the new value: ");
        Album_IO.Get(The_New_Album);
        Database.Modify(The_Album, To_Be => The_New_Album);
        Text_IO.Put_Line("Item has been modified");
```

```
when Command_IO.Start_Report =>
  Reports.Initialize(The_Report);
  Text_IO.Put_Line("Report has been Started");
when Command_IO.Find =>
  case The_Category is
    when Reports.Title =>
      Album_IO.Get(The_Title);
      Reports.Find(The_Title,The_Report);
    when Reports.Artist =>
      Album_IO.Get(The_Artist );
      Reports.Find(The_Artist ,The_Report);
    when Reports.Style =>
      Album_IO.Get(The_Style );
      Reports.Find(The_Style ,The_Report);
    when Reports.Year =>
      Album_IO.Get(The_Year );
      Reports.Find(The_Year ,The_Report);
    when Reports.Number =>
      Album_IO.Get(The_Number );
      Reports.Find(The_Number ,The_Report);
    when Reports.Song =>
      Album_IO.Get(The_Song);
      Reports.Find(The_Song,The_Report);
    when Reports.Length =>
      Album_IO.Get(The_Length);
      Reports.Find(The_Length,The_Report);
  end case;
  Report_IO.Put(Report.Length_Of(The_Report));
```

```
            when Command_IO.Sort =>
               Report_IO.Get_Sort(The_Category);
               Report_IO.Get(The_Order);
               Reports.Sort(The_Report, The_Category, The_Order);
               Text_IO.Put_Line("Sorting completed");
               Report_IO.Put(Reports.Length_Of(The_Report));
            when Command_IO.Display_Report =>
               Text_IO.Put_Line("Database report follows:");
               Reports.Initialize(The_Iterator, The_Report);
               Loop
                   exit when Reports.Is_Done(The_Iterator);
                   Album_IO.Put(Reports.Value_Of(The_Iterator));
                   Reports.Get_Next(The_Iterator);
               end loop;
               Report_IO.Put(Reports.Length_Of(The_Report));
         end case;
      end loop;
   End Process_Album_Database;
```

ولنعتبر نقطتين إضافيتين حول هذا التنفيذ البرمجي. أولاً، سيُدرك القارئ الحريص، بأنّه قد قدمنا تنفيذاً برمجياً ضعيف الجمالية. ولنعتبر ما يحدث إذا أدخل المستخدم معياراً غير صالح لـ Find (من المحتمل خطأ في كتابة *artist*). بالإضافة لذلك، من الممكن وجود إشكالات إذا استدعى المستخدم Modify، لتسجيلةٍ غير موجودة في قاعدة المعطيات. ومثلما درسنا في الفصل ٤، ففي مثل هذه الشروط، يمكن إبراز إستثناء. وبالتالي، سنؤجل دراسة تسهيلات معالجة الإستثناءات في ADA، وبالتفصيل حتى الفصل ١٥، إذ لم نستخدم برمجةً واقيةً لتبحث بهكذا مسائل لغاية الآن. وكنتيجةٍ لذلك، يتوقع حلّنا من المستخدم دخلاً تاماً - وهو شيء خطير.

ومن جهة أخرى، يسمح لنا هذا بتبسيط حلّنا، ولكن من جهةٍ ثانية، فإن هذا يجعل البرنامج الناتج غير قابلٍ للإستخدام. ومن أجل حاجات هذا الفصل، سنتابع إهمال معالجة شروط الإستثناءات. والحل الكامل لهذه المسألة، يجب أن يضاف له معالجة الإستثناءات كوسيلة برمجة واقية.

والنقطة الأخرى، تتعلق باستخدامنا للمكرّر. فلنركّز الآن على سلسلة التعليمات المتعلقة بمعالجة طلب المستخدم Display_Report. وتذكّر بأنّ هدف المكرّر، يتمثل بالسماح لنا بزيارة كل عنصرٍ من البنية. ففي هذه الحالة، نكرّر عبر الغرض The_Report، من أجل طباعة كل تسجيلة ألبومٍ في التقرير الذي بنيناه. ومن المهم أن يكون هذا التكرار غير هدام، ولا يغير حالة The_Report. لماذا؟ لأننا نريد متابعة معالجة هذا التقرير، مع توفير انطباعٍ وسيط.

ويوضح حلّنا الإستخدام التقليدي لمكرّر. فأولاً، يجب أن نستدعي Reports.Initialize، الذي، مثلما تمّ عرضه في الفصل ٧، يربط المكرّر (الغرض The_Iterator) بشكلٍ يشير للعناصر في غرض «التقرير» المحدد (The_Report). ثمّ، نطبّق «حلقة طالما» .. While loop، التي نتبعها حتى نستنفذ التكرار (المشار إليه عندما يتم تقييم Reports.Is_Done بـ True). وداخل هذه الحلقة، يمكننا تطبيق Reports.Value_Of لإعادة إيجاد معلومةٍ حول التسجيلة التي يشير إليها المكرّر حالياً. وبالإضافة لذلك، يجب أن نستدعي Reports.Get_Next لتقديم المكرّر إلى التسجيلة التالية.

ودعنا ننتقل إلى التنفيذ البرمجي لوحدةٍ أُخرى. حيث لا تحتاج الوحدة Albums لعملٍ إضافي، لأنه يوجد هنا لدينا فقط، مجموعة من تصريحات غير معلّبة؛ ولا توجد عمليات صريحة، بحاجةٍ للتنفيذ البرمجي. وعلى أية حال، فإن آلة الحالة-المجردة Database، تتطلب أن ننفذ برمجياً كل عملية تمّ عرضها في توصيفها. وللتذكّر، نعيد هذا التوصيف من الفصل ٧:

```
with Albums;
Package Database is
```

```
    Type Item is private;
    Procedure Create (The_Name : in String);
    Procedure Open   (The_Name : in String);
    Procedure Close;

    Procedure Add     (The_Albume : in Albums.Album);
    procedure Delete  (The_Albume : in Albums.Album);
    procedure Modify (The_Albume : in Albums.Album;
                      To_Be      : in Albums.Album);

    function Size return Natural;
    function Value_Of (The_Item : in Item) return Albums.Album;

    package Iterator is
        procedure Initialize;
        procedure Get_Next;
        function Value_Of  return Item;
        function Is_Done   return Boolean;
    end Iterator;

  Private
    type Node;
    type Item is access Node;
  end Database
```

وهذا يعطينا الرؤية الخارجية لتجريدنا. والآن، وفي هذه المرحلة، يجب أن نتوجه نحو الرؤية الداخلية. ويجب أن نأخذ بعين الإعتبار كيانين، هما : التمثيل الخارجي لقاعدة المعطيات،

و تمثيلها الداخلي. فبواسطة *الخارج*، نرجع لتسجيلات قاعدة المعطيات المخزّنة بطريقة مستمرة. ولحسن الحظ، توفر تسهيلات الدخل/الخرج بـ ADA التجريد الصحيح. وبسبب عدم امتلاكنا لأي قيدٍ على شكل تخزين قاعدة المعطيات،

بحيث يكون الشكل قابل للقراءة، فإننا سنستخدم Sequential_IO (بدلاً من Text_IO حسب الطريقة الموصوفة بالفصل ٤.

وبالتالي، يجب أن نُجري نسخة مؤقتةً عن Sequential_IO في جسم Database لتقديم تسهيلاتٍ لتخزين Albums.Album. ويجب أيضاً، أن نصرّح عن غرض File_Type. ويمكننا كتابة هيكل جسم الحزمة البرمجية هذه، كما يلي:

```
with Sequential_IO;

Package body Database is
      type Node is
            Record
               The_Album : Albums.Album;
               Next          : Item;
            end record;

    Package Album_IO is new Sequential_IO(Albums.Album);
    The_Items  : Item;
    The_File : Album_IO.File_Type;
    Has_Been_Modified : Boolean := False;
    Procedure Create (The_Name : in String) is separate;
    Procedure Open (The_Name : in String) is separate;
    Procedure Close is separate;
    Procedure Add (The_Album : in Albums.Album) is separate;
    Procedure Delete (The_Album : in Albums.Album) is separate;
    Procedure Modify (The_Album : in Albums.Album
                            To_Be : in Albums.Album) is separate;
    Function Size return Natural is separate;
    Function Value_Of (The_Item : in Item) return
     Albums.Album is separate;
    Package body Iterator is separate;
end Database;
```

وسنناقش تفاصيل معالجة الدخل/الخرج في الفصل ١٨. والآن، نحتاج فقط لعمليات بسيطة على الملفات Open, Close, Get, Put إذ أنّ دلالة كل عملية هي بديهية.

ولاحظ بأنه أيضاً، قد قدمنا الغرض المحلي Has_Been_Modified. وسنستخدم Has_Been_Modified ليساعدنا باتخاذ قرار، متى تمّ حفظ تغيرات The_File .

وبالتالي، عند إستدعائنا لـ Add أو Modify، سنعطي القيمة True للغرض Has_Been_Modify . وبالتالي، عند إغلاقنا لقاعدة المعطيات بـ Close ، فإننا سنكتب التغيرات على وحدة تخزين خارجية فقط، إذا أخذ Has_Been_Modified القيمة True . ويجب أن ندرك تماماً بأنّ وجود The_File وhas_Been_Modify في جسم Database، هما اللذان جعلا هذه الوحدة آلة حالة-مجردة. ولشكل هذين الغرضين حالة الحزمة البرمجية، لأنّه تمّ التصريح عنهما مباشرةً داخل الحزمة البرمجية - وليس داخل برنامجٍ جزئي. وسنناقش متضمنات هذه الطريقة في الفصل القادم.

ويعطينا The_File تجريداً للتمثيل الخارجي لقاعدة المعطيات، ولكن نحتاج أيضاً لتمثيلٍ داخلي. والحل البديهي لهذا، يتمثل باستخدام لائحةٍ مترابطة من الألبومات. ففي البدء، تكون هذه اللائحة فارغة، ولكن كلما نستدعي Add لتقديم عنصرٍ جديد، نحشر عقدةً جديدةً في اللائحة. وتتطلب Remove أن نتصفح هذه اللائحة، ونحذف العقدة التي تطابق معيارنا. وكذلك، تتطلب Modify أن نتصفح اللائحة، ولكن عندما نجد التطابق، نقوم ببساطة بتغيير تلك العقدة.

وإن النوع Item المطروح في توصيف Database، يشكل جزءاً من هذا التجريد. ولاحظ بأنّ الرؤية الخارجية لـ Item، تتمثل بنوعٍ معلب. ويجب أن نكمل هذا التصريح في القسم الخاص، حسب قواعد ADA، مثلما فعلنا في الفصل ٦، ويمكننا استخدام تصريح نوعٍ غير تام، لتأجيل تصريحنا للنوع Node حيث تشير له أغراض Items. وبالتالي، يتضمن القسم الخاص من Database التصريحات التالية:

في جسم Database، نُكمل تصريح Node، كما يلي:

```
type Node is
Record
```

```
   The_Album : Albums.Album;
   Next : Item;
End record;
```

لدينا هنا تعريف تراجعي، لأنّ أغراض Node نفسها، يمكن أن تؤشر إلى أغراض Node أُخر. وكما هو الحال بالنسبة لحالة The_File، سنصرح أيضاً عن غرضٍ محلي – The_Item من النوع Item – ليشير إلى التمثيل الداخلي لقاعدة المعطيات.

والآن، وقد أكملنا تصميمنا لبُنى المعطيات المستخدمة في Database، يمكننا المتابعة بالخوارزميات التي تعالج هذه البُنى. وتؤثر العمليات Cracte, Open, Close بشكل أساسي على الغرض The_File. ولهذا، تمّ بناء هذه العمليات على أساس العمليات الصالحة من إجراء نسخة مؤقتة من Sequential_IO باسم Album_IO . (لا تخلط هذه النسخة المؤقتة، مع الوحدة ذات الترجمة المنفصلة التي بنفس الإسم. ففي هذه الحالة، لدينا وحدة مصرحة محلياً، وهي مرئية فقط في جسم Database). وبشكل محدد، تستدعي Create بدورها Album_IO.Craete. وبشكل مشابه، تستدعي العملية Open Album_IO.Open، ولكن يجب أن تبني هذه الإجرائية لائحةً مترابطةً بدائية. والعملية Close لها الفعل المعاكس ،

فإذا تمّ تغير قاعدة المعطيات، فإنها تتصفح اللائحة المترابطة، وتكتب معلومات الألبوم على ملف. وبالتالي، يمكننا كتابة:

```
Separate (Database)
Procedure Create(The_Name : in String) is
Begin
    Album_IO.Create(The_File, Album_IO.In_File, The_Name);
End Create;

Separate (Database)
Procedure Open(The_Name : in String) is
    The_Album : Albums.Album;
Begin
    Album_IO.Open(The_File, Album_IO.In_File, The_Name);
```

```
    Loop
       exit when Album_IO.End_Of_File(The_File);
       Album_IO.Read(The_File, The_Album);
       The_Items := new Node'(The_Album => The_Album,
                              Next => The_Items);
    end loop;
End Open;


Separate (Database)
Procedure Close is
Begin
    if Has_Been_Modified then
       Album_IO.Reset(The_File, Album_IO.Out_File);
       Loop
          exit when The_Items=null;
          Album_IO.Write(The_File, The_Items.The_Album);
          The_Items := The_Items.Next;
       end loop;
       Has_Been_Modified := False;
    end if;
    Album_IO.Close(The_File);
End Close;
```

وتؤثر بشكل أساسي Add,Delette,Modify على التمثيل الداخلي لقاعدة المعطيات، وبشكل أكثر دقة فإن Add تتطلب ببساطة، حجز عقدةٍ جديدةٍ إلى اللائحة المترابطة The_Items

```
Separate (Database)
Procedure Add(The_Album : in Albums.Album) is
Begin
    The_Items:= new Node'(The_Album => The_Album,
                          Next => The_Items);
    Has_Been_Modified := True;
End Create;
```

وهنا أيضاً، لم نستخدم طريقةً برمجيةً ذات وقاية أفضل؛ ولنعتبر ما يحدث إذا أدخلنا معلومات ألبوم تضاعف تسجيلةً موجودةً مسبقاً في قاعدة المعطيات. ففي تنفيذٍ برمجيٍ أخير، نقدم خوارزميةً معقدةً قليلاً تستخدم الإستثناءات، ليتم تعريفها ضد هذه الإمكانية.

وتتطلب Delete وModify تصفح هذه اللائحة المترابطة، لإيجاد الألبوم الموافق. (وكما هو الحال مع Add، لم نحم أنفسنا من عدم إيجاد المطابقة). وتذكّر أنه حسب الرؤية الخارجية، يسمح لنا المكرّر، بتصفح كل العناصر في قاعدة المعطيات. ويمكننا إجراء الشيء ذاته من الرؤية الداخلية. وبالتالي، إن الحلقة الأولى في Delete وModify، هي واحدة من خلالها نعبر اللائحة The_Items نزولاً حتى نجد تطابقاً. وببساطة، تحذف Delete هذه العقدة (مع الإنتباه إلى الحالة التي تكون فيها العقدة المطابقة، هي العقدة الأولى في اللائحة). وببساطة تغير Modify معلومات الألبوم. وبالتالي، يمكننا كتابة:

```
Separate (Database)
Procedure Delete (The_Album : in Albums.Album) is
   previous_Item : Item;
   The_Iterator : Item := The_Items;
   function "="(Left,Right:in Albums.Album) return Boolean
         renames Albums."=";
Begin
   loop
      exit when The_Iterator = null
      or else The_Iterator.The_Album = the_Album;
      Previous_Item := The_Iterator;
      The_Iterator := The_Iterator.Next;
   end loop;
   if Previous_Item = null then
      The_Items := The_Items.Next;
   else
      Previous_Item.Next := The_Iterator.Next;
   end if;
```

```
      Has_Been_Modified := True;
   End Delete;
   Separate (Database)
   Procedure Modify (The_Album : in Albums.Album;
                              To_Be : in Albums.Album) is
      The_Item : Item;
      The_Iterator : Item := The_Items;
      function"="(Left,Right:in Albums.Album) return  Boolean
            renames Albums."=";
   Begin
      loop
         if The_Iterator = null then
            exit;
         elsif The_Iterator.The_Album = the_Album then
            The_Item := The_Iterator;
            exit;
         else
            The_Iterator := The_Iterator.Next;
         end if;
      end loop;
      The_Item.The_Album := The_Album;
      Has_Been_Modified := True;
   End Modify;
```

توجد نقطتان في الأسلوب يجب مناقشتهما. أولاً، أعطينا قيماً بدائية للغرض المكرّر The_Iterator عند تعريفنا له. وقد أجرينا ذلك دون الخوف من إبراز استثناء. وكوننا تخلصنا من الإستثناءات، فقد تمكنا من تبسيط أجسام الإجرائيات. ثمّ، أدرجنا تصريحاً لجعل عملية المساواة مرئيةً مباشرة. ومثلما سنناقش بتفصيل أكثر في الفصل التالي، لا تكون الكيانات المصدرة من حزمةٍ برمجيةٍ مرئيةٍ مباشرةً؛ ويجب تأهيل ذلك، إما بواسطة إسم الحزمة البرمجية، أو جعلها مرئيةً باستخدام عبارة use، أو إعادة تسمية التصريح. ونحن نفضل الطريقة الأخيرة. وبشكل مختلف عن عبارة use، فإن إعادة تسمية التصريح، تجعل فقط إسماً واحداً مرئياً مباشرة.

و المختاران الوحيدان في قاعدة المعطيـات همـا Size_Of وValue_Of. حيـث أن تنفيذهما البرمجي يكون مباشـرةً: و Size_Of تعـبر اللائحـة المترابطـة الممثلـة بــ The_Items، وتعد عدد العقد في طريقها. بينما تعيد Value_Of معلومـات عـن الألبـوم المرتبط بغرض من النوع Item :

```
Separate (Database)
function Size return Natural is
        The_Count : Natural := 0;
        The_Iterator : Item := The_Items;
    Begin
        Loop
            if The_Iterator = null then
                return The_Count;
            Else
                The_Count := The_Count + 1;
                The_Iterator := The_Iterator.Next;
            end if;
        end loop
    end Size;
    separate (Database)
    function Value_Of (The_Item : in Item) returns
                    Albums.Album is
    Begin
        return The_Item.The_Album;
    end Value_Of;
```

و المهمة الوحيدة الباقية، هـي التنفيـذ الـبرمجي لمكـرّر قـاعدة المعطيـات. ولا يملك جسم هـذه الحزمـة البرمجيـة المعششـة، أي نـوعٍ مـن الدهشـة، لأنّـه يـوازي إستخدامنا للمكرّر في Modify

و Delete، لذلك، يمكننا كتابة:

```
separate (Database)
package body Iterator is
```

```
The_Iterator : Item;
procedure Initialize is
Begin
   The_Iterator := The_Items;
end Initialize;
Procedure Get_Next is
Begin
   The_Iterator := The_Iterator.Next;
end Get_Next;
Function Value_Of return Item is
Begin
   Return The_Iterator;
end Value_Of;
Function Is_Done return Boolean is
Begin
   Return The_Iterator = null;
end Is_Done;
End Iterator;
```

وبهذا نُكمل التنفيذ البرمجي لـ Database؛ ولننتقل الآن إلى Reports، الذي يعتمد مباشرةً على هذا التجريد.

يوفر Reports نوع معطياتٍ مجرد، يُدعى Report. ومن وجهة الرؤية الداخلية، يمكن اعتبار Report، كمجموعة عناصر قاعدة معطيات. وفي أبسط تمثيل لها، نستطيع استخدام مصفوفة عناصرها من النوع Database.Item . وعلى أية حال، من غير الممكن لنا معرفة عدد العناصر التي يمكن أن تحتويها قاعدة المعطيات؛ وبالتالي، يجب أن نكون قادرين على تصريح أغراض تقرير، تحتوي على أعداد مختلفة من العناصر. وحل هذه المسألة يشابه الحل الذي فحصناه في الفصل ٦: حيث يمكننا استخدام مصفوفةٍ غير مقيدة، حيث عناصرها من النوع Database.Item. وعلى أية حال، فإن هذا جزء فقط من الحل. وبما أنّ غرض تقرير، طوال فترة حياته، يمكن ربطه بقواعد معطيات من أحجام مختلفة، يجب أن نضيف مستوى من العمل غير المباشر. وبالتالي، سنغلف هذه التسجيلة بمميز المصفوفة. ويمكننا التعبير عن قرارات التصميم هذه، بالتصريحات التالية:

```
Type Items is array (Positive range <>) of Database.Item;
Type Node (The_Size : Natural) is

   Record
     The_Items : Items(1..The_Size);
     The_Length : Natural := 0;
 end record;
```

ولكن يمكننا عمل أفضل فمن ذلك. من الرؤية الخارجية، سيكون أفضل إذا لم يعتبر المستخدم حجم قاعدة المعطيات، قبل التصريح عن غرض من النوع Node (لأنّه يجب أن يخبره). وبالتالي، سنضيف مستوى إضافياً من العملٍ غير المباشر: وبدلاً من تمثيل النوع Report على شكل Node، سنجعل Report مشيراً إلى Node. ومن أجل إعطاء قيمة بدائية لتقرير جديد، يجب أولاً حجز عقدة جديدة، والتي ستكون بمثابة مقيدٍ لحجم قاعدة المعطيات الحالية. وبالتالي، ففي القسم الخاص من Reports، يمكننا إكمال تصريح Report، كما يلي:

```
Type Node (The_Size : Natural);
Type Report is access Node;
```

ويمكننا إخفاء تمثيل النوع Node، بتوضيع تصريحه الكامل في جسم Reports، في نفس وقت التصريح عن النوع Items. وبما أنّه قد تمّ التصريح عن النوع Iterator في توصيف Reports أيضاً، فإنه يجب أن نكمل تصريحه أيضاً في القسم الخاص. وهنا، اخترنا لتمثيل هذا النوع كتسجيلة بمركبتين: مركبة تمثل غرض تسجيلة، والثانية، دليل يشير إلى المكان الحالي في لائحة العناصر، حيث يشير إليه المكرّر في ذلك الوقت. وبالتالي، يمكننا كتابة ما يلي:

```
type Iterator is
Record
   The_Report : Report;
   The_Index : Natural;
End record;
```

والآن، دعنا ننتقل إلى جسم Reports. فعلى سبيل المثال، يتطلب البناء Initializc، أن نبني تقريراً جديداً يحتوي على جميع العناصر في قاعدة المعطيات الحالية. وهنا، يجب أن نستخدم مكرر قاعدة معطيات. لعبور قاعدة المعطيات. وبدلاً من حفظ معلومات الألبوم نفسه، نحفظ أغراضاً من النوع Database.Item، كمؤشراتٍ لتسجيلات ألبوم شخصية.

وبالتالي، في جسم Initialize، نحجز أولاً غرضاً جديداً من النوع Node، والذي يكون مقيداً بحجم قاعدة المعطيات الحالية. ثم نستخدم المكرّر لزيارة كل عنصر:

```
Separate (Reports)
Procedure Initialize (The_Report : in out Report) is
function "=" (Left, Right : in Database.Item) return Boolean
        renames Database."=";
Begin
   The_Report := new Node(The_Size => Database.Size);
   Database.Iterator.Initialize;
   Loop
      exit when Database.Iterator.Is_Done;
      The_Report.The_Length := The_Report.The_Length + 1;
      The_Report.The_Items(The_Report.The_Length) :=
                              Database.Iterator.Value_Of;
      Database.Iterator.Get_Next;
End Initializle;
```

نجد بعد Initialize، العديد من الإجرائيات المحملة زائداً بإسم Find ، والتي تختار ببساطة عناصر من التقرير الحالي. وستعتمد إستراتيجيتنا على إدخال غرض محلي، The_Index، لزيارة كل عنصر في التقرير الحالي. وانطلاقاً من طرف المصفوفة The_Items، نتحقق من أنّ قيمة ذلك العنصر تطابق المعيار المحدد. فعلى سبيل المثال، إذا استدعينا Find لعنوان ألبوم، نرى فيما إذا كانت هذه القيمة تطابق مركب العنوان للعنصر المعين. وإذا حصلنا على تطابق، عندها لا نعمل شيئاً ونحتفظ بالعنصر

في التقرير. وإذا لم نحصل على تطابق، يجب أن نُبعد هـذا العنصـر. ولتحقيـق ذلـك، يمكننا استخدام إسناد شريحة مصفوفة، للكتابة فوق العنصر المسبب لذلك، ومـن ثـمّ، تقليص طول التقرير بمقدار ١. ويمكننا التعبير عن هذه الخوارزمية، بما يلي:

```
Separate ( Reports )
procedure Find (The_Title : in Albums.Title;
                In_The_Report : in out Report) is
   The_Index : Natural := In_The_Report.The_Length;
   function "=" (Left, Right : in Albums.Title) return
   Boolean
        renames Albums."=";

Begin
   while Th_Index > 0
   Loop
    if Database.Value_Of(In_The_Report.The_Items(The_Index))
    .The_Title
        The_Index := The_Index - 1;
     Else
        if The_Index < The_Report.The_Size then
            In_The_Report.The_Items
            (The_Index..In_The_Report.The_Length - 1)) :=
            In_The_Repot.The_Items((The_Index + 1 ) ..
             In_The_Report.The_Length);
        end if;
        The_Index := The_Index - 1;
        In_The_Report.The_Length := In_The_Report.
        The_Length - 1;
     end if;
 end Find;
```

ولـن نُضمّـن بقيـة التنفيـذ الـبرمجي لعمليـات Find، وذلـك لأنّ أجسـام هــذه الإجرائيات، مطابقة تماماً لجسم الإجرائية السابقة.

والإجرائية التالية، Sort، تستفيد من القرار الذي اتخذناه، بالحفاظ على التقرير، وكأنه مجموعة من عناصر تصف ألبومات، بدلاً من حفظ المعلومات كتسجيلاتٍ شخصية. وتتمثل الفائدة الأساسية، بأنّنا لا نُعرِّفُ إلا تكراراً واحداً من كل تسجيلة.

وقد اخترنا استخدام خوارزمية الفرز السريع. وبالتالي، يمكننا كتابة ما يلي:

```
separate ( Reports)
procedure Sort (The_Report : in out Report;
                By_Category : in Sort_Category;
                With_The_Order : in Order := Ascending) is
  function Incorrect_Order (Left : in Albums.Album;
  is separate;
  procedure Quicksort (Sort_Array : in out Items) is
     Front : Natural := Sort_Array'First;
     Back : Natural := Sort_Array'Last;
  procedure Exchange(First,Second: in out Database.Item) is
       Temporary : constant Database.Item := First;
     begin -- Exchange
       First := Second;
       Second := Temporary;
     end Exchange;
     pragama Inline (Exchange);
     procedure Partition is
       Mid_Point : constant Natural := (Front + Back) / 2;
       Mid_Value:constant Database.Item := Sort_Array Mid_Point);
     Begin
       Outer:
       Loop
          Loop
             exit When Incorrect_Order(Database.Value_Of
             (Sort_Array(Front)),
```

```
                                        Database.Value_Of(Mid_Value))
                          Or Front = Sort_Array'Last;
                     Front := Front - 1;
                  end loop
                  Loop
                     exit When Incorrect_Order(Database.Value_Of
                     (Mid_Value)),
                              Database.Value_Of(Sort_Array(Back))
                          Or Back = Sort_Array'First;
                     Back := Back - 1;
                  end loop
                     if Front <= Back then
                        if Front < Back then
                           Exchange (Sort_Array(Front),
                           Sort_Array(Back));
                        end if;
                        if Front /= Sort_Array'Last then
                           Front := Front + 1;
                        end if;
                        if Back /= Sort_Array'First thenend if;
                           Back := Back - 1;
                        end if;
                     end if;
                     exit Outer when (Front > Back)
                              or (Front = Sort_Array'Last
                              and Back = Sort_Array'First);
                end loop Outer;
              end Partition;
           begin -- Quicksort
              if Sort_Array'Length > 1 then
                Partition;
                if Sort_Array'First < Back then
                   Quicksort (Sort_Array(Sort_Array'First..Back));
                end if;
```

```
        if Front < Sort_Array'Last then
           Quicksort (Sort_Array (Front..Sort_Array'Last));
        end if;
     end if;
  end Quicksort;
begin -- Sort
  Quicksort(The_Report.The_Items(1..The_Report.The_Length));
end Sort;
```

يفرز Quicksort، مصفوفاتٍ عناصرها من النوع Item . وهكذا، لا تعمل إجرائيتنا الجزئية شيئاً، أكثر من اختيار العناصر الفعلية من التقرير، وتمررهما إلى Quicksort .

ويبدأ Quicksort بتقرير ما إذا كانت المصفوفة ذات طول كافٍ، ليُطلب ترتيبها (SortArray'Length > 1). وتتمثل الفكرة الأساسية من Quicksort، بتبديل مركباتٍ أبعد مايمكن. وتُقسم المصفوفة في المنتصف، وتبدل المركبات التي على الطرفين. ويتم تكرار هذه المعالجة على شرائح من المصفوفة، باستخدام التراجع.

ولاحظ كيف استخدمنا واصفات غرض في جسم Quicksort. فقد تمّ استدعاء Quicksort، على مصفوفات بأحجام مختلفة. وإنّه استخدام الواصفات، الذي سمح للبرنامج الجزئي أن يعمل على مصفوفة، من أي حجم.

وقد تمّ استخدام التابع الفرعي Incorrect_Order، لتعيين ما إذا كانت المركبات تحتاج للتبديل. ويجب تمرير Incorrect_Order إلى أغراض Albums.Album، التي يمكن أن نحصل عليها (اعتباراً من غرض من النوع Database.Items)، باستخدام مختار قاعدة المعطيات Value_Of. وبالإضافة لذلك، يعتمد Incorrect_Order على صنف الفرز (By_Category)، وترتيب (With_The_Order)، اللذين يمثلان شيئين عامين لهذا التابع الفرعي.

و هكذا يمكننا كتابة ما يلي :

```
Separate (Reports.Sort)
Function Incorrect_Order (Left  : in Albums.Album;
```

```
                    Right : in Albums.Album)
   return Boolean is
   use Albums;
Begin -- Incorrect_Order
   if With_The_Order = Descending then
      case By_Category is
when Title => return Left.The_Title < Right.The_Title;
when Artist => return Left.The_Artist < Right.The_Artist;
when Style => return Left.The_Style < Right.The_Style ;
         when Year => return Left.The_Year < Right.The_Year ;
         when Number => return Left.Number_Of_Songs<
          Right.Number_Of_Songs;
      end case;
   Else
      case By_Category is
         when Title => return Left.The_Title > Right.The_Title;
         when Artist=> return Left.The_Artist >Right.The_Artist;
         when Style => return Left.The_Style > Right.The_Style;
         when Year => return Left.The_Year > Right.The_Year ;
         when Number => return Left.Number_Of_Songs >
          Right.Number_Of_Songs;
      end case;
   end if;
End Incorrect_Order;
```

ويتضمن الفعل الرئيسي لـ Incorret_Order، تعليمتين Case ضخمتين، يتم من خلالهما تحديد علاقة الفحص الصحيحة لتطبيقها.

ويمثل Length_Of، المختار الوحيد المصدر من قبل Reports. وجسمه بسيط، لأن الحالة التي يعود لها، يمكن الوصول إليها مباشرةً:

```
Separate (Reports)
Function Length_Of (The_Report) return Natural is
Begin
   return The_Report.The_Length;
End Length_Of;
```

ويشبه المكرّر المصدر من Reports، لذلك المصدر من Database ،باستثناء أنّه في Reports، يوجد نوع معطياتٍ مجرد، وليست لآلة حالة-مجردة. وفي الواقع، إن التنفيذ البرمجي لكل عملية مكرّر يوازي العملية الموافقة في Database، باستثناء أنّ غرض المكرّر المستخدم يتم تمريره كمعامل، وبالتالي، يمكننا كتابة ما يلي:

```
Separate (Reports)
Procedure Initialize (The_Iterator : in out Iterator;
                      With_The_Report : in Report)is

   Begin
        The_Iterator := (The_Report => With_The_Report,
         The_Index => 1);
   end Initialize;
   __separate (Reports)__procedure Get_Next (The_Iterator : in out
   Iterator) is__begin__The_Iterator.The_Index := The_Iterator.The_Index
   + 1;__end Get_Next;____separate (Reports)__function Value_Of
   (The_Iterator : in Iterator) __return Albums.Album is__begin__return

   .The_Index));__end Value_Of;____separate (Reports)__function Is_Done
   (The_Iterator : in Iterator) return Boolean is
   begin
      return The_Iterator.The_Index >
      The_Iterator.The_Report.The_Length;
   end Value_Of;
```

وبهذا يكتمل تنفيذنا البرمجي لجسم Reports . أما التنفيذ البرمجي لبقية الحزم البرمجية Albums_IO, Report_IO, Command_IO فهو بسيط. ومثلما ناقشنا في الفصل ٧، تجمع هذه الحزم البرمجية، جميع العمليات النصية لتجريداتهن الموافقة. فعلى سبيل المثال، يتم بناء جسم الحزمة البرمجية Command_IO إعتبارا من Text_IO. وبالتالي، يستلزم جسم الإجرائية Get، استخدام الإجرائية Put، والإجرائية Get فقط. ويمكننا كتابة جسم الحزمة البرمجية Command_IO، كما يلي:

```
with Text_IO;
package body Command_IO is
   package IO is new Text_IO.Enumeration_IO(Command);
   procedure Space(Count : Positive) is
   Begin
     Text_IO.Put(String'(1..Count => ' '));
   end Space;
   procedure Get(The_Command : out Command) is
     prompt : constant String := "Possible commands are: ";
   Begin
     Text_IO.New_Line;
     Text_IO.Put(Prompt);
     for A_Command in Command
     Loop
     IO.Put(A_Command, Set => Text_IO.Lower_Case);
     if (Command'Pos(A_Command) mod 4) = 3 then
           Text_IO.New_Line;
           Space(Count => Prompt'Length);
         Else
           Space(Count => Command'Width + 1
                 - Command'Image(A_Command)'Length);
     end if;
        end loop;
     Text_IO.New_Line;
     Text_IO.Put("Waiting for command: ");
     IO.Get(The_Command);
     Text_IO.Skip_Line;
    endGet;
 end Command_IO;
```

وتطبع الإجرائية Get أسماء Command، ومن ثمّ تقرأ Command من دخل المستخدم. وتتم طباعة أربعة طلبات في كل سطر. ولاحظ كيف استخدمنا الواصفين

Pos وmod، لتحديد متى نكون قد طبعنا أربعة أسماء. ولاحظ أيضاً، استخدام الواصفات Width, Image, Length، من أجل وضع فراغاتٍ مناسبة لأسماء Command.

ولم ننه بعد، التنفيذ البرمجي لـ Report_IO, Album_IO، لأنهما يعتمدان فقط على تسهيلات الدخل/الخرج المسبقة التعريف في ADA. (لن نكمل دراستنا لهذه الموارد، حتى الفصل ١٨).





11

الحزم البرمجية
Packages

شكل الحزم البرمجية

الحزم البرمجية والأنواع الخاصة

تطبيقات الحزم البرمجية في ADA



يمكن أن يجمع المبرمج في ADA الموارد المترابطة في حزمة برمجية. فقد استخدم المثالان السابقان الحزم البرمجية في حلولهما. والآن يمكنكم الشعور بداهة بوظيفية هذه اللغة. وفي هذا الفصل، سندرس الحزم البرمجية عن قرب أكثر، وسنتفحص بنيتها وتطبيقاتها.

١١ ـ ١ ـ شكل الحزم البرمجية

(The Form of ADA Packages):

تعتبر الحزم البرمجية إحدى وحدات البرامج الأساسية في لغة ADA، إذ أنّه بواسطة الحزم البرمجية، يمكن تعليب مجموعة من «الكيانات» (Entities) المترابطة منطقياً، كما أنها تدعم مبادئ البرمجيات، في تجريد المعطيات وإخفاء المعلومات. وهنالك عدة طرق لاستخدام الحزم البرمجية، كالبرامج الجزئية. وإن الحزم البرمجية، تتألف من:

- توصيف الحزمة البرمجية (Specification Package).
- جسم الحزمة البرمجية (Body Package).

يمثل قسم توصيف الحزمة البرمجية واجهة تخاطب بين الحزمة البرمجية والمستثمر، إذ بواسطتها يتم تحديد أجزاء الحزمة البرمجية التي ستستخدم، وكيفية استخدامها.

وليس من المهم أن يعرف المستخدم كيف تم تنفيذ الحزمة البرمجية، وبالتالي، تهم الحزمة البرمجية المستخدم بقسمها المرئي فقط، وهو قسم التوصيف. فمثلاً، واجهة التخاطب بين الإنسان والسيارة، هي مقود القيادة، وعلبة السرعة، والمكبح. وجميع هذه الأقسام مرئية بالنسبة للسائق، الذي لا يهتم بكيفية عمل هذه الأقسام التي تعتبر تفصيلات التنفيذ. وبشكل مشابه لذلك في لغة ADA، هنالك بعض الأقسام المخفية في جسم الحزمة البرمجية، والتي لا تهم المستخدم، لكنها ضرورية لعمل البرنامج.

وهذه البنية، تدعم مبدأ الوحدوية، والتجريد، والمحلية، وإخفاء المعلومات. وبالطبع، فإن المبرمج الجيد، يستطيع تطبيق هذه المبادئ في بقية اللغات، حتى في FORTRAN أو لغة المجمع.

والفرق بين ADA وبقية اللغات، هو أنّ الحزم البرمجية بـ ADA، تحقق استخدام هذه المبادئ وتشجعها.

وإن قواعد لغة ADA، لا تسمح لمستخدم الحزمة البرمجية، بعمل أكثر مما هو مسموح به في توصيف الحزمة البرمجية. وإذا حاول المستخدم ذلك، فإنّ المترجم سيعطي خطأً دلالياً (Semantic Error).

وبما أنّه يمكن ترجمة قسم التوصيف والجسم لحزمة برمجية بشكل منفصل، فإنّه من المفيد جداً أن يتم خلق قسم التوصيف عند تصميم البرمجية، ومن ثم إضافة جسم الحزمة البرمجية فيما بعد.

وبالفعل، فإن هذا هو جوهر استخدام لغة ADA، والذي يتمثل بـ:

- أنها لغة تصميم للبرمجيات.
- تسمح بترجمة قسم توصيف الحزمة البرمجية، بشكل منفصل عن جسمها.
- إخفاء بعض المعلومات الهامة لتنفيذ البرنامج، وليست هامة للمستخدم.
- تساعد الحزم البرمجية في ADA المبرمج بالسيطرة على تعقيد الحلول البرمجية.

توصيف حزمة برمجية (Package Specifications):

إن توصيف الحزمة البرمجية، له الشكل التالي :

```
Package Some _Name is
......
......
End Some _Name;
```

ويمكن أن يقسم قسم التصريحات إلى قسمين رئيسيين، وهما:

- القسم المرئي.
- القسم الخاص.

والقسم المرئي، يصرّح عن المنابع التي يمكن أن تستخدم خارج الحزمة البرمجية، وبالتالي، يقال عن الحزمة بأنها ستصدر «كيانات» (Entities)، إذ أنّه يمكن تصدير أي عدد من العناصر المشكلة للحزمة البرمجية، مثلاً: الأغراض، والأنواع، والأنواع الجزئية، والبرامج الجزئية، والمهمات، والأعداد، والإستثناءات، والثوابت، وإعادة تسمية الفئات، وحتى الحزم البرمجية.

ومن المفضل أن يكون قسم توصيف الحزمة البرمجية صغيراً ويصدر نوعاً واحداً من القطع المنطقية.

وأما القسم الخاص، فيظهر فقط في الجزء الأخير من قسم التوصيف، ويكون مسبوقاً بالكلمة Private، وهو يشبه القسم المرئي، إذ أنّه يتألف من عناصر مصرح عنها، وهو لا يمكن أن يصدّر إلى خارج الحزمة البرمجية.

وبشكل عام، فإن الوحدة البرمجية، يمكن أن تستخدم المنابع المرئية من أية حزمة برمجية.

مثال: ليكن لدينا حزمة برمجية باسم Complex، وتوصيفها على الشكل التالي:

```
package complex is
type number is private;
procedure set (A_number       : out number;
              real_part      : in float;
              imaginary_part : in float);
function "+" (left,right : in number) return number;
function "-" (left,right : in number) return number;
function  real_part (A_number  : in number) return float;
function imaginary_part (A_number : in number) return float;
private
 type number is
  record
   real_part          : float;
```

```
  imaginary_part : float;
  end record;
end complex;
```

فالحزمة البرمجية Complex، مرئية بالنسبة للوحدة البرمجية P، إذا تم التصريح عنها في داخل P أو خارجها، ولم تكن مخفية بسبب تصريح آخر. مثلاً، ليكن Main برنامج رئيسي، ونريد أن نصرح في داخله عن عدة حزم برمجية، من ضمنها Complex، فيتم ذلك كما يلي:

```
Procedure Main is
Procedure  First is Begin .... End First;
Package Complex is  .... End Complex;
Package Body Complex is .... End Complex;
Procedure Second is ... ---- Another Declarative Item
Procedure Third is Begin .... End Third; --- A nested Procedure
 Begin .... End Second;
Begin  --- Main
 --- Sequence of Statements
End Main;
```

فمن خلال هذا المثال، يمكن القول بأنّ الحزمة البرمجية مرئية خلال البرنامج Main، اعتباراً من النقطة التي تمّت تسميته بها لأول مرة. وبالتالي، وبالإعتماد على قواعد الرؤية، فإنّ الإجرائية first، لا يمكن أن ترى الحزمة البرمجية Complex، بينما الإجرائية Second، والإجرائية Third، يمكن أن يستخدما منابع الحزمة البرمجية Complex.

وإن بعض المبرمجين، وربما أغلبيتهم، يفضلون استخدام عبارة With، من أجل رؤية منابع الحزمة البرمجية. فمثلاً، إن الحزمة البرمجية Complex، يمكن أن تترجم بشكل منفصل، ومن ثم تستخدم من قبل وحدة برمجية أُخرى، باستخدام العبارة Clause، كما يلي:

```
With Complex;
Procedure Main is
```

```
 Numb1,Numb2,Numb3:Complex.number;
 Fl1,Fl2: Float;
....
 Complex.set(Numb1,Fl1,Fl2);
  Numb3:=Complex."+"(Numb1,Numb2);
....
   Numb3:=Complex."-"(Numb1,Numb2);
....
    Fl1:=Complex.real_part(Numb1);
    Fl2:=Complex.imaginary_part(Numb2);
end Main;
```

فوفق هذا المثال، لاحظ أنّه قد تم في البدء استخدام ;With Complex. وهذا يعني، أنّ جميع المنابع المذكورة في توصيف الحزمة البرمجية Complex، سوف تصبح مرئيةً بالنسبة للبرنامج Main، ولكن استدعاءها، سيتم بذكر إسم الحزمة البرمجية، قبل أي عنصر من الحزمة، متبوعاً بـ ".". ولاحظ أنّه عندما أردنا التصريح عن الغرضين Numb1, Numb2، على أنهما من النوع number، المعرّف في الحزمة البرمجية Complex، تمّ ذلك على الشكل ;Numb1,Numb2:Complex.number. وأيضاً، عندما استخدمنا البرامج الجزئية المعرّفة ضمن الحزمة البرمجية Complex، قد تم ذلك بنفس الطريقة التي صرّحنا بها عن الغرضين Numb1, Numb2.

فمثلاً، إن استدعاء:

- الإجرائية set، يتم كمايلي: Complex.set (Numb1,Fl1,Fl2).

- التابع الفرعي "+"، يتم كمايلي:

Numb3:=Complex."+"(Numb1,Numb2)

وهكذا ....

ويمكن التصريح عن بعض الأغراض، واستدعاء بعض البرامج الجزئية، دون ذكر إسم الحزمة البرمجية، التي تحتوي هذه الأغراض والبرامج الجزئية. ويتم ذلك باستخدام تعليمة الرؤية المباشرة Use، عند تحديد الحزمة البرمجية. فمثلاً من أجل

استخدام منابع الحزمة البرمجية Complex دون ذكر اسمها يتم ذلك بإضافة Use Complex; بعد ذكر With Complex.

وبالتالي، يمكن التصريح عن Numb1,Numb2، على أنهما من النوع Number، المعرّف بالحزمة البرمجية Complex، كما يلي:

Numb1,Numb2:number;

وأيضاً، يتم استدعاء الإجرائية set، كما يلي: **set(Numb1,Fl1,Fl2);**

بينما يتم استدعاء التابع الفرعي "+"، كمايلي: Numb3:=Numb1+Numb2; ، أيضاً يتم استدعاء التابع الفرعي "-" بنفس الطريقة.

بينما يتم استدعاء التابع الفرعي real_part، والتابع الفرعي imaginary_part بنفس الطريقة التي تم فيها استدعاء الإجرائية set. مثلاً Fl1:=Real_Part(Numb1); .

ويمكن استدعاء أكثر من حزمة برمجية بنفس الوقت، من أجل وحدة برمجية معينة، وذلك كما يلي:

With Package1,Package2;With Package3;...;With PackageN;

ونفس الشيء، بالنسبة لـ Use .

وعند استخدام Use، يمكن أن يتجاهل المستثمر وجودها، ويصرّح عن بعض الأغراض

و كأنّه لم يتم استخدام Use. كذلك الأمر، بالنسبة لاستدعاء البرامج الفرعية.

ولكن يفضل استخدام With دون Use، وذلك من أجل أن تكون الوحدة البرمجية أكثر قابليةً للقراءة، وخاصةً عندما تكون عدد الحزم البرمجية المستدعاة كثيرة، واحتمال وجود عدة أنواع مختلفة البنية تحت نفس الإسم، في عدة حزم برمجية مختلفة.

ولاحظ أنّه قد تم إجراء تحميل زائد على بعض التوابع الفرعية المعرّفة ضمن المكتبة القياسية على مختلف الأنواع البسيطة المعرفة بلغة ADA، مثل التابع "+" والتابع "-".

جسم الحزمة البرمجية (Package Bodies):

ويأخذ جسم الحزمة البرمجية الشكل التالي:

```
Package Body Some _Name is
.....
End Some _Name;
```

حيث:

- Some _Name، يمثل إسم الحزمة البرمجية، ويجب أن يطابق تماماً إسم توظيف الحزمة البرمجية.

ويجب أن يكون لكل توصيف حزمة برمجية، جسم حزمة برمجية، باستثناء توصيف الحزم التي تحتوي الأنواع والأغراض. وبالتالي، في هذه الحالة، فإن جسم الحزمة البرمجية يكون خيارياً.

والعناصر التي تتواجد داخل جسم الحزمة البرمجية، لا يمكن الوصول إليها، ولا يمكن رؤيتها خارج الحزمة البرمجية، وهذا ما يدعم مبدأ إخفاء المعلومات.

وإن جسم الحزمة البرمجية، له شكل مشابه تقريباً لشكل البرامج الفرعية. إذ أنّه يتألف من قسم تصريحات، متبوعاً بكتلة خيارية، مع سلسلة من التعليمات، وبعض الإستثناءات الخيارية، مثل جسم البرامج الجزئية، أي تصريح محلي أو وحدة برمجية محلية، يمكن أن تدخل ضمن جسم الحزمة البرمجية.

وعند بناء جسم الحزمة البرمجية، يجب أن يتم بناء قسم التصريحات أولاً، ومن ثمّ سلسلة التعليمات.

مثال:

فيما يلي، سنستعرض مثالاً عن مجموعة عمليات حسابية بسيطة على الأعداد العقدية، نوضح من خلاله توصيف الحزمة البرمجية، وجسمها.

توصيف للحزمة البرمجية Complex :

```
package complex is
type number is private;
procedure set (A_number      : out number;
```

```
            real_part      : in float;
            imaginary_part : in float);
function "+" (left,right : in number) return number;
function "-" (left,right : in number) return number;
function  real_part (A_number  : in number) return float;
function imaginary_part (A_number : in number) return float;
private
 type number is
  record
   real_part         : float;
   imaginary_part : float;
  end record;
end complex;
```

جسم الحزمة البرمجية Complex:

إن جسم الحزمة البرمجية Complex، له الشكل التالي:

```
package body complex is
 procedure set (a_number       :  out number;
             real_part       : in float;
             imaginary_part : in float) is
 begin
  A_number := (real_part,imaginary_part);
 end set;
-----------------
 function "+" (left,right : in number) return number is
  begin
   return (left.real_part+right.real_part,
         left.imaginary_part+right.imaginary_part);
  end "+";
-----------------
 function "-" (left,right : in number) return number is
  begin
   return (left.real_part-right.real_part,
         left.imaginary_part-right.imaginary_part);
```

```
 end "-";
----------------
function real_part(A_number : in number) return float is
 begin
  return A_number.real_part;
 end real_part;
----------------
function imaginary_part (A_number : in number) return float is
 begin
  return A_number.imaginary_part;
 end imaginary_part;
end complex;
```

ومن خلال ذلك، نستنتج أنّه لا يوجد قسم تصريحات خاص بالحزمة البرمجية Complex، كما أنّه من خلال جسم الحزمة البرمجية، تمّ تحديد جسم كل برنامج جزئي، تمّ توصيفه في توصيف الحزمة البرمجية Complex.

مثال عن استدعاء الحزمة البرمجية «العدد العقدي» Complex:

فيمايلي، برنامج رئيسي يتم من خلاله استدعاء عناصر الحزمة البرمجية Complex، وبعض البرامج الجزئية المحلية، الخاصة بالبرنامج الرئيسي :

```
with complex,text_io;
 procedure Ex_Complex is
  first,second:complex.number;
  a,b: float;
  package complex_io is new text_io.float_io(float);
------------------------
  procedure read_complex (A_number : out complex.number) is
  a,b:float;
 begin
  text_io.put("Enter the real  part: ");
  complex_io.get(a);
  text_io.put("Enter the imaginary part: ");
  complex_io.get(b);
```

```
 complex.set(a_number,a,b);
end read_complex;
--------------------------
procedure write_complex(A_number: in complex.number) is
begin
 text_io.put("the  real part is equal to: ");
 complex_io.put(complex.real_part(a_number),aft=>2,exp=>0);
 text_io.new_line;
 text_io.put("The imaginary_part is equal to: ");
   complex_io.put(complex.imaginary_part(a_number),aft=>2,
                                               exp=>0);
 text_io.new_line;
end write_complex;
-------------------------
begin   ----------- Ex_Complex
 read_complex(first);
 write_complex(first);
 read_complex(second);
 write_complex(second);
 text_io.put("---------------");
 text_io.put("After the substraction second from
first");text_io.new_line;
 first:= complex."-"(first,second);
 text_io.put("the real part of first is ");
 complex_io.put(complex.real_part(first),aft=>2,exp=>0);
 text_io.new_line;
 text_io.put("the imaginary part of first is ");
 complex_io.put(complex.imaginary_part(first),aft=>2,exp=>0);
 text_io.new_line;
end Ex_Complex;
```

١١ ـ ٢ ـ الحزم البرمجية والأنواع الخاصة

(Packages&Private Types) :

فيما سبق، قد نوهنا بأن الحزم البرمجية يمكن أن تجبر على مبدأ التجريد. فغالباً ما يريد المبرمج خلق غرض، بحيث تكون الخواص المنطقية محفوظة خارج الحزمة البرمجية، بينما التفاصيل البنيوية غير هامة (Irrelevant).

وهذا مرتبط بشكل أولي بالأنواع الخاصة، حيث الأنماط الخاصة، لا يمكن تعريفها إلا في القسم المرئي من الحزم البرمجية.

ويوجد صنفان من الأنواع الخاصة، وهما :

- الأنواع الخاصة البسيطة (Simple).
- الأنواع الخاصة المحدودة (Limited).

فبالنسبة للأنواع الخاصة البسيطة، فإن المعلومة الوحيدة الممكن استخدامها خارج الحزمة البرمجية، هي المعطاة في القسم المرئي من الحزمة البرمجية، التي تم فيها التصريح عن تلك الأنواع.والعمليات الوحيدة الجاهزة للأغراض لنوع خاص، هي البرامج الجزئية المصرح عنها في القسم المرئي، بالإضافة لعملية الإسناد، وفحص عملية المساواة وعدمها.

وتطبق للنوع الخاص المحدود، نفس القواعد المطبقة على النوع الخاص، ماعدا عملية الإسناد، وفحص المساواة وعدمها، حيث لا يمكن استخدامها خارج الحزمة البرمجية.

فإذا احتوت حزمة برمجية على تعريف أنواع خاصة، فإنه يجب أن يحتوي قسم التوصيف على قسم خاص، يتمم تعريف النوع. ويمكن للقسم الخاص، أن يحتوي أشياء أخرى غير الأنواع الخاصة. أيضاً، يمكن لحزمة برمجية، وبدون أنواع خاصة، أن تحتوي على قسم خاص.

وفي قسم التوصيف، لا يمكننا فقط تعريف أنواع خاصة، بل يمكن أيضاً التصريح عن ثوابت من أنواع خاصة. ومثال ذلك، يمكن خلق حزمة برمجية تعطي

Password، والتحقق من مدى صحتها. فمجرد أن نخلق Password، نريد أن نعطيها قيمة بدائية Null_Password، التي يتم التصريح عنها كثابت، كما يلي:

```
Package Manager is
  type  Password is Private;
  Null_Password : Constant Password;
  Function Get Return Password;
  Function Is_Valid(A_Password : in Password) return
 Boolean;
  Private
     Type Password is  range 0..7_000;
      Null_Password : Constant Password :=0;
End Manager;
```

لاحظ أنّه قد صرحنا Password على أنّها Private، ولم تُحدد على أنها Limited Private، وذلك من أجل أن تسمح للمستثمر بعملية إسناد غرض من هذا النوع، لغرض آخر.

ويمكن استخدام Null_Password خارج الحزمة البرمجية، على الشكل التالي:

```
With Manager;
Use   Manager;
My_Password : Password :=Null_Password;
```

وإن الأنواع الخاصة، والأنواع الخاصة المحدودة، تسمح للمبرمجين بتطبيق تحكم كامل على عمليات ممكنٌ تطبيقها لأنواع مصدّرة. وهذه الملامح، تعتبر نافعةً عند خلق معطيات مجردة. وهناك تفاعل دقيق بين بعض القواعد الخاصة بالنوع Acces، والأنواع الخاصة. فقواعد ADA، تتطلب بأن تكون الأنواع الخاصة المقدمة في قسم توصيف الحزمة البرمجية، بواسطة تصريح من نوع كامل في القسم الخاص، وبمعنى آخر، عند الإنتهاء من توصيف الحزمة البرمجية، يجب أن نقدم التنفيذ الكامل لكل الأنواع الخاصة.

وعلى أية حال، في ADA، يمكن إتمام تعريف الأنواع الخاصة في جسم الحزمة البرمجية. ومثال ذلك، يمكن أن نمثل النوع Password، على شكل نوع مؤشر في قسم توصيف الحزمة البرمجية. لذلك، يمكن أن نكتب ما يلي:

```
Type Node;
Type Password is access Node;
```

وفي جسم الحزمة البرمجية Manager، يمكن أن نتمم التصريح عن النوع Node، كمايلي:

```
Type Node is range 0..7_000;
```

١١ - ٣ - تطبيقات الحزم البرمجية

(ADA Applications for Packages) :

يمكن استخدام الحزم البرمجية في لغة ADA، في التطبيقات الأربعة التالية:

- تسمية مجموعة من التصريحات.
- تجميع عدة وحدات برمجية، مرتبطة فيما بينها.
- تجريد أنواع المعطيات.
- تجريد حالة الآلة.

تسمية مجموعة من التصريحات (Named Collections of Declarations):

إن أحد استخدامات الحزم البرمجية البسيطة، هو التجميع المنطقي للأغراض والأنواع. ويفيد هذا التطبيق في الصيانة، بتجزئة المعطيات المشتركة، والأغراض، والأنواع المشتركة، وتوضيع تعريفها في مكان واحد.

ويمكن لهذه التعاريف أن تستخدم من قبل أية وحدة برمجية أخرى. فعلى سبيل المثال: في نظام يتطلب نموذجاً للكرة الأرضية، كبرنامج ملاحة، أو تطبيق خرائطي. فمن المهم جداً الإحتفاظ بمجموعة من الثوابت الهامة في مكان واحد، لاستخدامها من قبل بعض الوحدات البرمجية الأخرى. يمكن ذلك، باستخدام الحزم البرمجية كما يلي:

```
Package Metric_Earth_Constants is
  Equatorial_Radius  Constant := 6_378.145;      -- km
  Gravation_Constant:Constant:=3.986_012E5; --  km**3/sec**2
  Speed_Unit  : Constant := 7.905_368_28;    --  km/sec
  Time_Unit   : Constant := 806.811_874_4; -- sec
End Metric_Earth_Constants;
```

ففي هذا المثال، لا توجد أية وحدة برمجية في قسم توصيف الحزمة البرمجية. لذلك، يمكننا إهمال جسم الحزمة البرمجية.

وكتطبيق آخر للحزم البرمجية، يمكن أن نجمع مجموعةً من الأنواع المترابطة منطقياً. وفي نظام يستخدم التاريخ، فإنه من المفيد جداً وضع أنواع اليوم، والشهر، والسنة، في حزمة برمجية واحدة، على الشكل التالي:

```
Package Date_Information is
  Type Day_Name is
   (Monday,Tuesday, Wednesday, Thursday, Friday, Saturdy, Sunday);
  Type Day_Value is range 1..31;
  Type Month_Name is
     (January, February, March, April, May, June,
      July,August,September,October,November,December);
Type  Year_Vlue is Range 0..Integer'Last;
End Date_Information;
```

فوفق هذا التوصف، لا داعي لوجود جسم الحزمة البرمجية Date_Information.

ووفق هذه الحزمة البرمجية، تمّ تحديد أسماء الأسبوع كنوع مرقم وهي تتألف من يوم الاثنين وحتى الأحد. كما حددنا قيمة اليوم، وهي تنتمي للمجال [31 1]. وأيضاً فإن أسماء الأشهر قد تمّ تحديدها كنوع مرقم. وأخيراً، فإن قيمة السنة قد تم تحديدها بعدد طبيعي مقبول تمثيله ضمن لغة ADA.

تجميع عدة وحدات برمجية، مرتبطة فيما بينها:

في الأمثلة السابقة، قد تمّ تجميع معطيات مرتبطة منطقياً فيما بينها. ومن الممكن أيضاً تجميع وحدات برمجية مثل البرامج الجزئية، والمهمات، أو حزم برمجية أخرى.

أمثلة:

١ ـ بما أنّ ADA لا تمتلك توابع مثلثية معرّفة مسبقاً بمكتبتها الأصلية، فيمكن تعريف حزمة برمجية خاصة بذلك، كما يلي:

```
Package Transcendental_Functions is
   Function Cos (Angle : in float)  return float;
   Function Sin  (Angle : in float)  return float;
   Function  Tan (Angle : in float)  return  float;
End Transcendental_Functions;
```

فوفق هذا التوصيف، نلاحظ أنّه لا غنى عن جسم الحزمة البرمجية Transcendental_Functions، الذي يتألف من أقسام خاصة بجسم الحزمة البرمجية، ولا يمكن تصديرها. بالإضافة إلى جسم كل برنامج جزئي من البرامج الجزئية الموصّفة في توصيف الحزمة البرمجية Transcendental_Functions، حيث يتم زرع جسم الحزم البرمجية، والذي يستخدم سلاسل مثلثية لحساب القيم، والتي تأخذ الشكل التالي:

```
 Package Body Transcendental_Functions is
  Series_Length : Constant :=5;
 function Odd (Index : in Integer) return Boolean is
     --- Return True If Index has an Odd Value
  Begin
   return (Index rem 2)/=0;
  End Odd;
 ---
 function Factorial (Value : in Positive) Return Positive is
   --- Determine Value! Using A Recursive Function
   Begin
    if Value =1 then
      return 1;
     else
      return Value*Factorial(Value-1);
    End if;
   End Factorial;
 -----
  Function Term (Angle : in Float; Power : in Integer;
```

```
                 Number : in Integer) Return Float is
 --- Calculate a term of the Trigonometric series
Begin
 if Odd(Number) Then
    return -(Angle**Power)/Float(Factorial(Power));
   else
    return +(Angle**Power)/Float(Factorial(Power));
 end if;

Function Cos (Angle : in Float) return Float is
 -- Calculate The Cosine Of Angle
 Answer : Float;
 Power  : Integer;
Begin
 Answer :=1.0;
 For Index in Reverse 1..Series_Length
  Loop
   Power :=Index+Index;
   Answer := Answer+Term(Angle,Power,Index);
  End Loop;
 Return Answer;
End Cos;
------
function Sin (Angle : in float) return float is
 --- Calculate The Sine Of Angle
 Answer : float;
 Power  : Integer;
 Begin
  Answer := Angle;
  For Index in revese 1..Series_Length
   Loop
    Power := Index+Index+1;
    Answer := Answer + Term(Angle,Power,Index);
   End Loop;
   Return Answer;
 End Sin;
```

```
function Tan(Angle : in Float) return Float is
  --- Calculate The Tangent Of Angle
Begin
  Return Sin(Angle)/Cos(Angle);
End Tan;
End Transcendental_Functions;
```

لاحظ أنه من خلال تعريف جسم الحزمة البرمجية Transcendental_Functios، قد تمّ تعريف الثابت Series_Length، الذي يحدد طول السلسلة المثلثية، إذ أنّ التصريح عنها كثابت، يسهل على المبرمج تغيرها عند اللزوم. ولاحظ أنّه قد تمّ التصريح عن البرامج الجزئية Odd, Factorial, Term، لتكون برامج جزئية مساعدة، وخاصة بهذه الوحدة البرمجية، إذ أنّ هذه البرامج الجزئية، تصبح مخفيةً بالنسبة لبقية الوحدات والحزم البرمجية الأخرى.

٢ ـ إن الحزم البرمجية البيانية، تشرح تطبيقاً آخراً للحزم البرمجية بلغة ADA كتجميع للبرامج الجزئية، بحيث تكون البرامج الجزئية (Rotate, Scalc, Translate) خوارزميات بيانية مشتركة، ويمكن أن تقدم للمستخدم بالتوصيف التالي:

```
Package Two_D_Transform is
 Type Coordinate is
  Record
   X,Y : Float;
  End Record;
Procedure Rotate(Point :in out Coordinate; Angle : in Float);
Procedure Scale (Point : in out Coordinate; X,Y  : in Float);
Procedure Translate(Point : in out Coordinate; X,Y : n loat);
End Two_D_Transform;
```

فوفق هذا التوصيف للحزمة البرمجية Two_D_Transform، يمكننا القول بأنّ النوع Coordinate غير معلّب (Unencapsulated)، وأنه غير خاص، وغير خاص محدود.

وهذه الحزمة البرمجية، تحتاج للحزمة البرمجية Transcendental_Functions، التي تمّ تعريفها سابقاً. وبالتالي، فإن جسم الكتلة البرمجية Two_D_Transform، يأخذ الشكل التالي:

```
With Transcendental_Functions;
Use   Transcendental_Functions;
Package Body Two_D_Transform is
Procedure Rotate(Point: in out Coordinate; Angle:in Float) is
   --- Rotate the Point By Angle Radians About The Origin
    Temporary : Coordinate := Point;
   Begin
     Point := (X=> (Temporary.X*Cos(Angle))+
                 Temporary.Y*Sin(Angle)),
             Y=> -(Temporary.X*Sin(Angle))+
                 (Temporary.Y*Cos(Angle)));
   End Rotate;
Procedure Scale(Point :in out Coordinate; X,Y :in Float) is
 --- Scale The Point By A Factor Of X And Y

  Begin
    Point:=(X=>Point.X*X,Y=> Point.Y*Y);
  End Scale;
Procedure Translate(Point:in out Coordinate; X,Y:in Float) is
 --- Translate The Point By A Distance Of X And Y
 Begin
    Point:=(X=>Point.X+X,Y=> Point.Y+Y);
 End Translate;
 End Two_D_Transform;
```

أنواع المعطيات المجردة (Abstract Data Types):

في لغة ADA، يعتبر خلق نوع من المعطيات المجردة، آليةً جيدةً وميزةً تتصف بها عن بقية لغات البرمجة عالية المستوى. وهذا ما لاحظناه في عدة أماكن من قسم التوصيف لبعض الحزم البرمجية، إذ أنّ التجريد ليس إجبارياً في حال النوع غير الخاص وبنيته المرئية.

وفي حال التجريد المنطقي الإجباري، يجب استخدام الأنواع الخاصة. ففي حال الحزمة البرمجية Two_D_Transform، يمكننا التصريح عن الـ Coordinate، على أنها من النوع الخاص، ومن ثم نقل قسم التنفيذ إلى القسم الخاص في توصيف

الحزمة البرمجية. وبالتالي، يصبح شكل توصيف الحزمة البرمجية Two_D_Transform، على الشكل التالي:

```
Package Two_D_Transform is
 Type Coordinate is Private;
 Procedure Rotate(Point:in out Coordinate; Angle : in Float);
 Procedure Scale (Point : in out Coordinate; X,Y : in Float);
 Procedure Translate(Point: in out Coordinate;X,Y: in Float);
 Private
 Type Coordinate is
  Record
   X,Y : Float;
  End Record;
End Two_D_Transform;
```

وهنالك مثال آخر، نبين فيه استخداماً مميزاً في تعريف نوع خاص. ومثال ذلك ما يلي:

```
Package Pacakge_Example is
 Type Type_One(Size: Positive) is Private;
 Procedure Procedure_One .....
 Procedure Procedure_Two ...
.....
 Private
 ....
 Type List is Array(Positive range<>) of integer;
 Type Type_One(Size: Positive) is
    Record
     The_Items:List(1..Size);
     The_Back : Natural;
    End Record;
End Package_Example;
```

فوفق هذا المثال، قد تمّ تحديد Type_One على أنّه نوع خاص، ويتحدد بدلالة المميز Size. وبعد ذلك، تمّ تحديد قسم التوصيف للبرامج الجزئية، التي ستُصدر إلى خارج هذه الحزمة البرمجية. وبعد ذلك، تمّ تحديد القسم الخاص بالحزمة البرمجية، إذ أنّه تم تعريف List، على أنّه شعاع من الأعداد الطبيعية غير محدودة الطول. ومن

ثمّ النوع Type_One بدلالة Size، إذ أنّه يمثل تسجيلةً مركبتها الأولى مؤلفة من List بطول Size، ومركبته الثانية من النوع Natural.

حالة الآلة (أوتومات) المجردة (Abstract-State Machines) :

الأوتومات، هو كيان (Entity) له حالات معرَّفة بشكل جيد، وعمليات انتقال من حالة إلى أخرى. وأيضاً، إن الغاية الأساسية، هو اكتشاف الحالة الآنية للآلة.

ويمكن أن نمثل الأوتومات، بمجرد علب سوداء بسيطة من الأغراض. ويمكن للمستخدم أن يتفاعل مع العلبة، بواسطة برامج جزئية (إجرائيات)، أو يفحص خواص العلبة بواسطة توابع. وهذه الخواص، هي حالة الأوتومات. وكل فعل، يمكن أن يغير حالته الراهنة.

وباستخدام مفهوم الحزم البرمجية لتمثيل الأوتومات المجرد، لا داعي لتصدير أنواع، أو عموماً أغراض.

وبالتالي، تبدو الأوتومات المجرد، كأنواع معطيات مجردة، أو مجموعة بسيطة من الوحدات البرمجية. وعلى أية حال، فإن الفرق الأساسي، هو أن استخدام الحزم البرمجية كأوتومات مجرد، والتي تحفظ معلومات عن الحالة في جسم الحزمة البرمجية، يمكن تطبيق عمليات بناء (إجرائيات) أو اختيار (توابع فرعية) على الأوتومات، بدلاً من البرامج الجزئية في الحزم البرمجية العادية. مثلاً، إن الحزمة البرمجية Transcendental_Functions، التي تمّ التصريح عنها سابقاً، كانت مؤلفة من عدة برامج جزئية، ولكن استدعاء أحد البرامج الجزئية، لا يغير الحالة.

وبالحقيقة، فإن كل البرامج الجزئية كانت مستقلة بشكل تام، ولا يؤثر وجود أي منها على الأخرى.

ولكن من أجل الأوتومات، فإن تطبيق العمليات، يؤدي بشكل فعال إلى تغيير الحالة. وعلى سبيل المثال، يمكن تجريد «فرن» (Furnace) للمستخدم، على الشكل التالي:

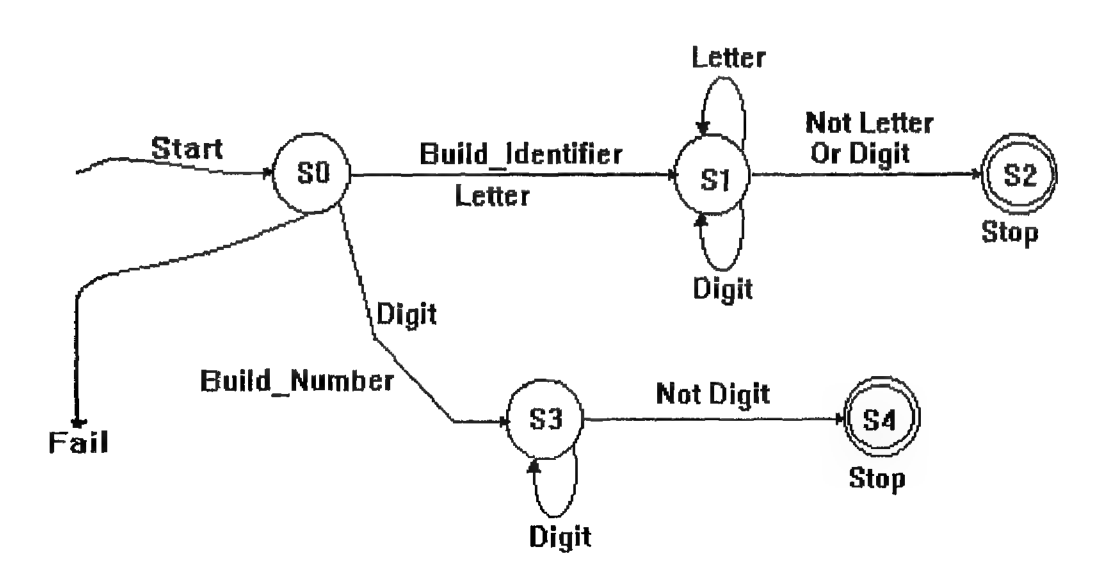
```
Package Furnace is
  Procedure Set(Tempreture : in Float);
  Procedure Shut_Down;
  Function Is_Running  Return Boolean;
  FuTemperature_Is  Return Float;
  Overtemp  : exception;
End Furnace;
```

ويمكن تصنيف العمليات المطبقة كعمليات بناء (إجرائيات)، أو عمليات اختيار (توابع فرعية). إن مستخدم الحزمة البرمجية، يمكنه فقط تطبيق البرامج الجزئية الأربعة المعطاة، من أجل تغيير حالة الفرن أو تحديد درجة الحرارة الآنية. ومن الواضح وجود فرق في ترتيب استدعاء البرامج الجزئية الأربعة السابقة، وهذا عائد للمعطيات المحفوظة. وبما أنّه لا توجد أنواع معطيات ولا أغراض للتصدير، فإنّ هذه الحزمة البرمجية، تعرّف فقط، غرضاً واحداً.

وبما أنّ توصيف الحزمة البرمجية يحتوي عدة إجرائيات، وبالتالي، يجب أن يحدد جسم الحزمة البرمجية بشكل كامل فيما بعد.

وكمثال نموذجي آخر على الأوتومات، يمكن أن نأخذ محلل المفردات (Lexical Analyzer)،

و الذي بواسطته يمكن تعيين وتصنيف سلسلة محارف. وتستخدم بعض الآلات كأدوات ضرورية للمترجمات، ونظم الإتصالات. وعلى سبيل المثال، وكجزء من مترجم، يمكن أن نخلق الأوتومات الذي يحدد الأسماء (identifiers) أو الأعداد. وإن مخطط انتقال الحالة لمحلل المفردات، له الشكل ١١ ـ ١:



الشكل ١١ ـ ١. أوتومات لمحلل المفردات.

ومثلما نلاحظ في المخطط، نجد أنّ الآلة في البدء، هي في حالة إقلاع. وعند استقبال أول رمز، تنقل إلى حالة Build_Identifiers إذا كان محرفاً أبجدياً، وإلى حالة Build_Number إذا كان خانة عشرية. وتبقى الآلة في كل من تلك الحالات التي انتقلت إليها، مادامت تستقبل محرفاً أبجدياً أو خانةً عشرية (من أجل Build_Identifier) أو خانة عشرية فقط (من أجل Build_Number). وفي حال استقبال رمز غير ذلك، يتم الإنتقال إلى حالة Stop، وفي تلك اللحظة، يتم تحديد المفردة (Token).

ويمكننا تعريف واجهة التخاطب للأوتومات المجرد، كما يلي:

```
Package Lexical_Analyzer is
  Type Token is (None, Invalid, Identifier, Number);
  Procedure Set_Start_State;
  Procedure Recive_Symbol(A_Character : in Character);
  Function Value Return Token;
End Lexical_Analyzer;
```

لاحظ أننا حددنا الشروط المقبولة، كنوع مرقم محرفي. وإن مستخدم هذه الحزمة البرمجية، يمكنه المتابعة بإرسال رموز إلى حالة الأوتومات، دون انتظار فحص قبول المفردة. فإذا أردنا، يمكننا تمرير المعلومات المقبولة بسهولة، كمعاملات (Parameters) من خلال Recive_Symbol. ولاحظ عند تعريف النوع Token، أنه تضمن Invalid التي تستخدم في حال كون الرمز الأول ليس محرفاً أبجدياً، وليس خانةً عشرية. وأيضاً، ضمّنا العملية Set_Start_State، إذ يمكن للمستخدم بواسطتها، أن يعيد الأوتومات إلى حالته البدائية.

ويمكن تنفيذ كل ما سبق ذكره برمجياً، كما يلي:

```
Package Body Lexical_Analyzer is
  Type State is (Start,Build_Identifier,Build_Number,Stop);
   Present_State : State :=Start;
   The Result : Token := None;
   SubType Alpha is Character Range 'A'..'Z';
   SubType Digit is Character Range 'O'..'9';
```

```
Procedure Set_Start_State is
 --- Initialize State machine
Begin
 Present_State:=Start;
 The_Result:=None;
End Set_Start_State;
Procedure Recive_Symbol(A_Character : in Character) is
 ---- Accept a symbol from the transmitter
Begin
 Case Present_State is
  When Start =>
    Case A_Character is
      When Alpha => Present_State := Build_Identifier;
      When Digit => Present_State := Build_Number;
      When Others => The Result := Invalid;
    End Case;
   When Build_Identifier =>
      Case A_Character is
        When Alpha | Digit => Null;
        When Others => Present_State := Stop;
                The_Result := Identifier;
      End Case;
   When Build_Number =>
      Case  A_Character is
        When Digit => Null;
        When Others => Present_State := Stop;
               The_Result := number;
      End Case;
   When Stop => Null;
 End Case;
End Recive_Symbol;
Function Value Return Token is
Begin
End Value;
End Lexical_Analyzer;
```

وبهذا تنتهي برمجة جسم الحزمة البرمجية Lexical_Analyzer، وهو عبارة عن محلل مفردات بسيط.

ويمكننا تغير الأوتومات، وبالتالي، الحزمة البرمجية، من أجل قبول مفردات أخرى، موصفة في الأوتومات.

وتستمد الحزم البرمجية قوتها، من قدرتها على تعليب كيان منطقي، ومن ثم تجبر على التجريد الموافق.



12

الوحدات البرمجية المولّدة
Generic Program Units

شكل الوحدات البرمجية المولّدة بلغة ADA

المعاملات المولّدة

تطبيقات ADA للوحدات البرمجية المولّدة



حتى الآن، رأينا كيفية تجزئة النظام إلى عدة وحدات برمجية. ونجد بشكل عام، أن هناك وحدات برمجية أو برامج جزئية، لها أهداف متشابهة. مثال ذلك، أنه يمكن أن يوجد برنامج جزئي يفرز أعداداً صحيحة، وبرنامج جزئي آخر، يفرز سلسلة من المحارف. ويمكن تنفيذ ذلك بسهولة، بواسطة برنامج جزئي واحد بلغة ADA، حتى لو اختلفت أبعاد سلسلة الأعداد وسلسلة المحارف.

ومثال آخر، هو أن خوارزمية تبديل أي عنصرين من نفس النوع ثابتة. فيمكن كتابة برنامج جزئي يبدل عنصرين من نوع مجرد، واستدعائه حسب الحاجة، من أجل تبديل عنصرين من النوع الصحيح أو النوع الحقيقي، أو محرفين،.. ويمكن قياس الكثير على ذلك .

١٢ ـ ١ ـ شكل الوحدات البرمجية المولّدة بلغة ADA:

(The Form of ADA Generic Program Units):

تسمح ADA بخلق حزم برمجية، وبرامج جزئية مولّدة. ومثلما سنرى لاحقاً، فإنه للحصول على مهمات مولّدة، يجب أن تكون المهمات ضمن حزمة برمجية.

والوحدات البرمجية المولّدة تعرف وحدة قالبية مع معاملات مولّدة التي تسهّل تكيُّف الوحدة القالبية وفقاً لاحتياجات خاصة في زمن الترجمة (التحويل Translation).

وبما أنّ الوحدات المولّدة هي فقط مرجعية، فهي ليست تنفيذية، ولا يمكن بالتالي، استخدامها بشكل مباشر. ويجب في البدء خلق نسخة من الوحدة المولّدة. وبعد ذلك، يمكننا استخدام البرنامج الجزئي الموافق أو الوحدة البرمجية الموافقة، وكأنها وحدة برمجية طبيعية. ويمكن تشبيه الوحدات البرمجية المولّدة بالنسبة للحزم البرمجية والبرامج الجزئية، كمثل الأنواع بالنسبة للأغراض.

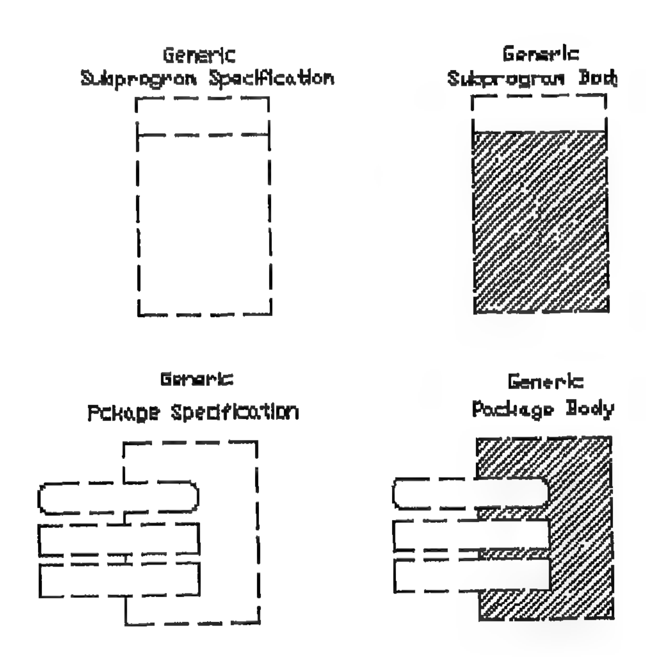
تعريف التوليد (Generic Definition) :

لخلق وحدة برمجية مولّدة، يمكن وببساطة، أخذ توصيف حزمة برمجية أو توصيف برنامج جزئي، ومن ثم إضافة السابقة Generic له، والتي تعرّف جميع المعاملات المولّدة.

مثال:

يمكن خلق البرنامج الجزئي النظامي التالي، لتبديل عنصرين من النوع الصحيح:

```
Procedure Integer_Exchange(First,Second : in out Integer) is
  Temporary : Constant Integer := First;
Begin
  First     :=Second;
  Second := Temporary;
End Integer_Exchange;
```



الشكل ١٢ - ١. رموز وحدات البرامج المولّدة في ADA.

وإذا أردنا إجراء تبديل عنصرين من نوع آخر، فليس من الضروري خلق برنامج جزئي جديد من أجل كل حالة، إذ أنّه في كل حالة، تكون الخوارزمية هي نفسها، ولكن الذي يتغير هو نوع الغرضين الذين سيتم تطبيق الخوارزمية عليهما. ومن أجل تحقيق ذلك باستخدام Generic، يتم ذلك في قسم توصيف الإجرائية Exchange، كما يلي:

```
generic
    type Element is private;
procedure Exchange(First,Second : in out Element);
```

ويمكننا الآن كتابة جسم الإجرائية Exchange، فيصبح كما يلي:

```
Procedure Exchange(First,Second : in out Element) is
 Temporary : Constant Element := First;
Begin
 First    := Second;
Second := Temporary;
End Exchange;
```

لاحظ أنّ صيغة الخوارزمية المطبقة على جسم هذا البرنامج الجزئي، مطابقة تماماً لصيغة الخوارزمية المطبقة على البرنامج الجزئي Integer_Exchange، باستثناء أنواع المعطيات. وفي قسم التوليد، تمّ التصريح عن Element، وهذا ما يشكل المعاملات لقالب البرنامج الجزئي.

ففي كل نقطة تمّ فيها استخدام إسم النوع Integer من قبل، بدّلنا المعامل Element. وكما سنرى لاحقاً، إنّ لغة ADA تزودنا بتسهيلات توليد عامة، وهذا ما يسمح بالتصريح عن عدة صفوف من المعاملات المولّدة، بالإضافة للأنواع، متضمناً القيم، والأغراض وبرامج جزئية أُخرى.

وبما أنّ الوحدة البرمجية المولّدة، يمكن أن تكون وحدة ترجمة، فيمكننا إخضاع البرنامج الجزئي Exchange للترجمة. وقواعد لغة ADA تتطلب بأن تكون أسماء الوحدات وحيدة، وغير مكرّرة ضمن منطقة تصريح محددة، ولا يمكن إجراء تحميل زائد على أسماء الوحدات البرمجية المولّدة.

وعلى أية حال، فإن الترجمة لا يمكن أن تكون إلا جزئية، طالما أننا لم نربط المعاملات الصورية المولّدة مع المعاملات الحقيقية من بعض الوحدات البرمجية الأب. وبما أنّ Exchange يعرّف فقط، قالب وحدة برمجية نظامية، فلا يمكننا استخدام تسهيلاته مباشرةً، إذ أنّه في البدء، يجب خلق نسخة من الوحدة البرمجية المولّدة.

مثال: إن خلق وحدة برمجية بدءاً من وحدة غير مولّدة، جاهزة مسبقاً، هو أمر سهل. فمثلاً كنا قد عرفنا QUEUE.QUEUES من أجل عناصر Integer . فمن أجل جعل هذه الوحدة مولّدة، نضيف إلى قسم التوصيف للحزمة البرمجية البدائية، جزءاً مولّداً يعطي الإسم الصوري للمعامل، وهذا الإسم، يستعمل لترميز نوع العناصر. وبالتالي، فإن توصيف الحزمة البرمجية المولّدة يكون Queues تعالج عدة عمليات على الأرتال لنوع مجرد من المعطيات، ليتم تطبيقها على عدة أنواع مختلفة، إذ أنّه يتم تحديد النوع عند الحاجة، ويأخذ قسم توصيف الحزمة البرمجية Queues، الشكل التالي:

```
Generic
  type Item is Private;
package Oueues is
 type Queue (Size : Positive) is limited private;
 procedure Clear (The_Queue : in out Queue);
 procedure Add(The_Item:in Item;On_The_Queue:in out Queue);
 procedure Remove(The_Item : out Item;
                From_The_Queue : in out Queue);
 function Length_Of(The_Queue : in Queue) return Natural;
 Overflow,Underflow : exception;
Private
 type List is Array(Positive range <>) Of Item;
 type Queue(Size : Positive) is
   record
     The_Items : List(1..Size);
     The_Back  : Natural :=0;
  end record;
End Queues;
```

بينما جسم الحزمة البرمجية Queues، يأخذ الشكل التالي:

```
Package Body Queues is
 Procedure Clear(The_Queue : in out Queue) is
  begin
   The_Queue.The_Back:=0;
  end Clear;
```

```
Procedure Add(The_Item:in Item; To_The_Queue:in out Queue) is
  begin
   To_The_Queue.The_Items(To_The_Queue.The_Back+1)
                                    := The_Item;
   To_The_Queue.The_Back:=To_The_Queue.The_Back+1;
   exception
     when Constraint_Error => raise Overflow;
  end Add;
 Procedur Remove(The_Item : out Item;
            From_The_Queue : in out Queue) is
   Back : Natural renames From_The_Queue.The_Back;
  begin
   if Back=0 then
     raise Underflow;
     else
       The_Item := From_The_Queue.The_Items(1);
       Back := Back-1;
       From_The_Queue.The_Items(1..Back):=
                From_The_Queue.The_Items(2..Back+1);
   end if;
  end Remove;

 Function Length_Of(The_Queue : in Queue) is
  begin
   return The_Queue.The_Back;
  end Length_Of;
End Queues;
```

النسخ المولد (Generic Instantiation) :

يعرّف الـ Instantiation، على أنّه معالجة خلق نسخة من حزمة برمجية مولّدة أو برنامج جزئي مولد. وجزء من معالجة النسخ، يرتكز على تقديم معرّف سيسمي الوحدة البرمجية النظامية. ويجب أيضاً ربط المعاملات المولّدة مع المعاملات الحقيقية (كربط المعاملات الحقيقية، بالمعاملات الصورية، بالبرامج الجزئية). ونضيف ما هو ضروري في الوحدة البرمجية القالبية، وذلك بإجراء تبديل واحد لواحد، لكل تكرارٍ من

المعاملات الصورية المولّدة، مع المعاملات الحقيقية الموافقة. ومـن تلك النقطـة، يتـم الإستمرار في بناء القالب المكتمـل بهـذا الشـكل، وكأنـه تعريـف حزمـة برمجيـة، أو برنامج جزئي.

وبالرجوع إلى مثال القسم السابق، يمكـن التصريـح عـن عـدة نسـخ مـن الوحـدة Exchange، كما يلي:

```
Procedure Integer_Exchange is new Exchange(Integer);
Procedure Float_Exchange is new Exchange(Element => Float);
Procedure Character_Exchange is new
            Exchange (Element => Character);
```

لاحظ أنّ شكل تعريـف المعـاملات الحقيقيـة المولّـدة، مشـابه تمامـاً لاسـتدعاء البرامج الجزئية.

أما بالنسبة للحزمة البرمجية Queues، فيتم تعريف حزم برمجية مولّـدة منهـا، من أجل أنواع مختلفة، كما يلي:

```
package Integer_Queues   is new Queues(Item => Integer);
package Character_Queues is new Queues(Item => Character);
```

لاحظ أننا استخدمنا تجميع المعاملات المسمّاة، من أجل جعـل التصريـح أكـثر قابليةً للقراءة. ولاحظ أيضاً، بأنّ النسـخة المولّـدة، يمكـن تنشـيطها بـأيّ نقطـة يتـم التصريح فيها عن برنامج جزئي أو حزمة برمجية. وبـالطبع إن إسـم الوحـدة المولّـدة، يجب أن يكون مرئياً في نقطة النسخ.

والآن، وبمـا أننـا خلقنـا نسـخاً مـن الوحـدة المولّـدة Exchange، فإنـه يمكننـا استخدام الوحدات البرمجية النظامية مباشرةً، كما في الأمثلة التالية:

```
My_Character,Your_Character : Character;
Character_Exchange(My_Characte,Your_Character);
```

وكما سنناقش لاحقاً، فإن الحزمة البرمجية المسبقة التعريف Tcxt_IO، تصـدر عدة وحدات برمجية مولّدة ومتراكبة.

ومثل الحزم البرمجية والبرامج الجزئية، فإن الوحدات المولّدة ليست بالضرورة وحدات مكتبية، بل يمكن أن تكون متراكبة داخل وحدات برمجية أخرى. فمثلاً، إن الحزمة البرمجية Text_IO.Integer_IO، تعتبر حزمةً برمجيةً مولّدة متراكبة، تزودنا بتسهيلات الـ I/O من أجل الأنواع الصحيحة. وقبل استخدامنا لهذه المنابع، يجب نسخ هذا المولّد المتراكب، مع النوع الموافق. ومثال ذلك:

```
type Index is range 0..1_000_000;
package Index_IO is new Text_IO.Integer_IO(Index);
package Positive_IO is new Text_IO.Integer_IO(Positive);
```

ففي السطر الثاني من هذا المثال، تمّت عملية نسخ الحزمة البرمجية المولّدة Text_IO.Integer_IO، لتتعامل فقط مع الأغراض التي من النوع Index .

بينما في السطر الثالث من هذا المثال، تمّت عملية نسخ الحزمة البرمجية المولّدة Text_IO.Integer_IO، لتتعامل فقط، مع الأغراض التي من النوع Positive، المعرّف مسبقاً.

وفي أمثلة سابقة، وفي أكثر من مكان، تمّ نسخ عدة حزم برمجية مولّدة، وذلك حسب الحاجة.

١٢ - ٢ - المعاملات المولّدة (Generic Parameters):

سنفحص في هذا القسم، عدة صفوفٍ من المعاملات المولّدة. إذ أنّ مختلف المعاملات، يمكن أن تصنف فيما يلي:

- معاملات الأنواع المولّدة (Generic type parameters).
- معاملات القيم والأغراض المولّدة (Generic value and object parameters).
- معاملات البرامج الجزئية المولّدة (Generic subprogram parameters).

ومن الضروري ملاحظة أنّ المعاملات المولّدة ليست ساكنة (Static) على الإطلاق، لذلك لا يمكن استخدامها في تعابير ساكنة.

وفيما يلي، شرح مفصل لكل صنف من أصناف المعاملات.

معاملات الأنواع المولّدة (Generic type parameters):

مثلما ناقشنا في الحزم البرمجية والبرامج الجزئية، فإن لغة ADA تسمح لنا بتمرير قيم الأغراض فقط، إلى وحدات برمجية معينة. وعلى أيّة حال، ومثلما رأينا في القسم السابق، من الضروري في بعض الأحيان، تمرير أنواع معطيات كمعاملات. وبالرغم من أن قواعد الأنواع القوية، لا تسمح لنا بتمرير أنواع للوحدات البرمجية النظامية في زمن التنفيذ، فمن المكن الحصول على نفس التأثير من خلال استخدام معاملات الأنواع المولّدة. كما أنه يوجد في البرامج الجزئية، معاملات فعلية (حقيقية) ومعاملات صورية، كما يوجد أيضاً، معاملات مولّدة فعلية وصورية.

وبما أنّ ADA تزودنا ببنى أنواع كثيرة وغنية، فإنه يمكن استخدام عدداً كبيراً من معاملات الأنواع المولّدة المختلفة. وعلى كل حال ، فإن المعامل الصوري، يجب أن يربط مع معامل فعلي موافق. وتحقق ADA هذه القاعدة، لتضمن لنا الأمان الذي توفره أنواع المعطيات، حتى بين الوحدات خلال الترجمة المنفصلة . فإذا كان النوعان غير متوافقين، فإنّ المترجم سيشير إلى خطأ دلالي (إذا كان من الممكن اكتشاف الخطأ في زمن الترجمة)، أو أن النظام سيشير إلى خطأ من النوع Constraint_Error (إذا تمّ اكتشاف الخطأ أثناء التنفيذ).

واللائحة التالية، تلخص شكل جميع معاملات الأنواع المولّدة، وتعرّف أيّ المعاملات الفعلية تكون متوافقة. وإن الأسماء التي تبدأ بأحرف كبيرة، ليست كلمات محفوظة (ليست من أصل اللغة)، وليس لها أي دلالة خاصة في لغة ADA، إذ أنّها تمثل فقط، وظيفتها البنيوية.

```
type General_Purpose is limited private;
   -- matches any data type
type Element is private;
   -- matches any data type that permits assignment and
   -- tests for (in)equality
type Link is access Some_Object;
   -- matches any access type designating the same type
   -- of object
```

```
type Enumeration is (<>);
   -- matches any discrete (integer and enumeration)type
type Integer_Element is range <>;
   -- matches any integer type
type Fixed_Element is delta <>;
   -- matches any fixed_point type
type Float_Element is digits <>;
   -- matches any floting_point type
type Constrained_Array is array(Some_Index) of Some_Element;
   -- matches any constrained array of the same
   -- dimensions, index types, and type of components
type Unconstrained_Array is array(Some_Type range <>)
      of Some_Element;
   -- matches any unconstrained array of the same
   -- dimensions, index types, and type of components
```

وفي جسم الوحدة المولّدة، فإن العمليات والخواص المتاحة على الأنواع الفعلية، متاحة أيضاً على الأنواع المولّدة. مثال ذلك، إذا استخدمنا Float_Element، كما تمّ التصريح عنه قبل قليل، فإن جسم الوحدة المولّدة، يمكنه استخدام كل عوامل النقطة العائمة المعرفة سابقاً. وبشكل مماثل، من اجل الأنواع المولّدة الخاصة فقط، الإسناد والفحص من أجل المساواة وعدمها، هي المتاحة. ومن أجل الأنواع المولّدة والخاصة والمحدودة، وليس هناك أي عامل متاح لها، باستثناء العوامل المعرّفة كمعاملات برامج جزئية مولّدة.

وبعكس معاملات البرامج الجزئية، فإن ADA، تسمح بتعريف ارتباطات بين معاملات الأنواع المولّدة. مثلاً، يمكننا تعريف البرنامج الجزئي Sort ذي الأهداف العامة، كما يلي:

```
Generic
  type Element is <>;
  type List is array (Positive range <>) of Element;
 Procedure Sort(Table : in out List);
```

وفي قسم التصريح هذا، عن البرنامج الجزئي المولّد، فـإن مكونـات الشـعاع المولّد List، هي أيضاً معاملات صورية مولّدة. فيمكننا إجــراء نسخ عـن البرنـامج الجزئي المولّد Sort :

```
type Color is (Black, Green, Red, Blue, White);
type Color_Table is array(Positive range <>) of Color;
procedure Sort_Color is new Sort(Element => Color,
                                 List => Color_Table);
```

معاملات القيم والأغراض المولّدة

(Generic value and object parameters):

تسمح لغـة أيضـاً ADA بتعريـف قيـم وأغـراض، كمعـاملات صوريــة مولــدة. والتصريح لمعاملات كهذه، يأخذ شكل التصريـح عـن متغـيرات، وذلـك بإضافـة كلمـة المفتاح in (من أجل معامل القيمة) أو كلمة المفتاح in out (من أجل معـامل الغـرض). والنموذج out، غير ممكن من أجل معـاملات الأغـراض المولّـدة. وعنـد الحاجـة، فـإن معامل نوع مولّد مصرح عنه مسبقاً، يمكن استخدامه كنوع لمعامل غرض أو قيمة مولّدة.

وعند استخدام معامل قيمة مولّدة، يجـب ربـط القيمـة الصوريـة، مـع ثـابت أو متغير من نفـس النـوع. وإن لغـة ADA تعـالج القيمـة كثـابت مـن أجـل بقيـة الوحـدة البرمجية. والفائدة من هذا الصف من المعاملات المولّدة، هي أنه يمكننا تسريع الوحدة البرمجية، بتزويد خوارزميات أو أغراض مجردة بحدود ثابتة. وتسـمح هـذه السـهولة، بخلق لغة ضخمة عالية المستوى.

فعلى سبيل المثال، يمكن أن نعـرّف الحزمـة البرمجيـة التاليـة، والتـي تعـرّف طرفي مولّد، كنوع معطيات مجرد:

```
Generic
 Rows     : in Integer := 24;
 Columns : in Integer := 80;
 Package Terminal is ....
```

وبعد ذلك، يمكننا خلق عدة نسخ مؤقتـة مـن هـذه الحزمـة البرمجيـة المولـدة، كمايلي :

```
Package Micro_Terminal is new Terminal(24,40);
Package Word_Terminal is new Terminal(Rows  =>66,
                      Columns => 132);
Package Program_Terminal is new Terminal;
```

لاحظ كيف استخدمنا عدة أشكال من النسخ المؤقتة المولّدة. ففي المثال الأول، استخدمنا تجميع المعاملات الموضعي. وفي المثال الثاني، استخدمنا تجميع المعاملات المسمّاة، ليصبح التصريح أكثر وضوحاً. وفي المثال الثالث، استخدمنا القيم الفرضية (Default Values) للمعاملات المولّدة، فلم يكن ضرورياً حشر أي معامل فعلي في المثال الثالث.

وللتصريح عن معامل غرض مولد، نستخدم كلمة المفتاح in out. ويجب أن نربط المعاملات الصورية بمتغيرات من نفس النوع. وإن تأثير النسخة، هو إعادة تسمية المعامل الصوري إلى معامل فعلي. وكنتيجة لذلك، فإن السهولة تسمح لنا بوصول غرض عام إلى برنامج جزئي أو حزمة برمجية، والتي لا تكون بشكل طبيعي، طرقاً برمجية عملية. وإن معامل غرض مولّد، لا يمكن إعطاؤه قيمة تعبير فرضية (Default Expression Value). وإن معاملات القيم المولّدة، والأنواع المميزة، يمكن أن تستخدم للوصول إلى غايات مشابهة.

فعلى سبيل المثال، في بدء هذا الفصل ، في الحزمة البرمجية المولّدة Queues، قد جعلنا الأنواع بدلالة المعامل المميز Size. وبنفس الطريقة، يمكن إعطاء كل غرض من النوع Queue، طولاً أعظمياً وحيداً. فما هي الفائدة التي نجنيها، إذا كتبنا الترميز التالي:

```
Generic
  type Item is private;
  The_Size : in Positive;
package Queues is ....
```

إن معامل القيمة المولد The_Size، يمكن استخدامه لتعين الحجم الأعظمي لجميع الأغراض المصرح عنها من النوع Queue. وبهذه الطريقة، نحصل على حل وسط باستخدام المميز، بحيث أن كل غرض يمكن أن يعين بدلالة قيم مختلفة، ولكن

باستخدام معاملات القيم المولّدة. وإن جميع الأغراض من نفس النسخ المؤقتة، تشارك بنفس الصفات.

معاملات البرامج الجزئية المولّدة (Generic subprogram parameters):

تسمح لنا لغة ADA أيضاً، بتمرير البرامج الجزئية كمعاملات إلى وحدات مولّدة. فعلى سبيل المثال، إن الحزمة البرمجية Terminal التي تمّ تعريفها سابقاً، يمكن أن تحتاج لإرسال ولاستقبال معطيات، باستخدام عدة اتفاقيات (Protocols)، بالإعتماد على نوع الطرفي الفيزيائي المرتبط به. وإننا ننصح باستخدام الحزمة البرمجية المولّدة، لأنّ تجريدنا المنطقي لكل طرفي، هو بشكل افتراضي، شبيه بالطريقة التي نشكل بها الشاشة. وعندها يمكننا معالجة Send وrecive، كمعاملات برامج جزئية مولّدة.

```
Generic
  Rows      : in Integer := 24;
  Columns : in Integer := 80;
  With Procedure Send  (Value : in Character);
  With Procedure Recive(Value : out Character);
 package Terminal is ....
```

وعندما نجري نسخةً مؤقتةً عن Terminal، يجب تزويد برنامجين جزئيين، بحيث يكون تصريحهما متوافق مع Send وRecive. (والمقصود بكلمة متوافق هنا، بأنّ البرامج الجزئية، تأخذ معاملات من نفس النوع، والنموذج، والترتيب، ومبنية بشكل البرنامج الجزئي الصوري، بالإضافة إلى قيمة معادة متطابقة في حال الوظيفة).

مثال :

```
procedure Micro_Send   (Value : in Character) is ...
procedure Micro_Recive(Value : out Character) is ...
Package Micro_Terminal is new
  Terminal (Rows => 24, Columns => 40,
     Send => Micro_Send, Recive =>Micro_Recive);
```

وتسمح ADA أيضاً، بالتصريح عن قيم فرضية (Declaration of defaults) من أجل معاملات البرامج الجزئية المولّدة، باستخدام أحد الشكلين التاليين:

في الشكل الأول، نستخدم كلمة المفتاح is، متبوعة بإسم البرنامج الجزئي الإفتراضي (Default). على سبيل المثال:

```
With Text_IO;
Generic
   Rows      : in Integer := 24;
   Columns : in Integer := 80;
   With procedure Send  (Value : in Character) is Text_IO.Put;
   With procedure Recive(Value : out Character) is Text_IO.Get;
Package Terminal is ...
```

وفي الشكل الثاني، نستخدم كلمة المفتاح is، متبوعة بالرمز <> . في هذه الحالة، يمكننا إهمال المعامل الفعلي الموافق، إذا كان البرنامج الجزئي بنفس الإسم، ويملك قسم توصيف متوافق مع توصيف معامل البرنامج الجزئي المولّد، والمرئي في نقطة النسخ المؤقت. وعلى سبيل المثال، لنعتبر التصريح التالي:

```
Generic
  Rows     : in Integer := 24;
  Columns : in Integer := 80;
  With procedure Send  (Value : in Character) is <>;
  With procedure Recive(Value : out Character) is <>;
 package Terminal is ...
```

ويمكننا النسخ عن الحزمة البرمجية، إذا كان هنالك برنامجان جزئيان Send وRecive، متوافقين ومرئيين في تلك النقطة. (متوافق، تعني أنّ التصريح عن البرامج الجزئية متطابق، دون اعتبار أسماء المعاملات الصورية، أو الحضور، أو القيم الإفتراضية).

مثال:

```
procedure Send  (Value : in Character);
procedure Recive(Value : out Character);
package Dumb_Terminal is new Terminal;
```

وبما أن البرنامجين الجزئيين المحلين Send وRecive مرئيان في النقطة التي طلبنا فيها النسخ، فلا يمكننا تزويد أيّ معامل فعلي، للربط مع البرامج الجزئية الصورية المولّدة.

والشكلان الأخيران لمعاملات البرامج الجزئية الإفتراضية، تشكل تسهيلات للمبرمجين، بالرغم من أنها تقلّص وضوح النسخة المولّدة في بعض الأحيان. فمن المفضل بشكل عام، والعملي، جعل معامل البرنامج الجزئي الفعلي كإسم صريح.

١٢ ـ ٣ ـ تطبيقات ADA للوحدات البرمجية المولّدة:

(Applications for ADA Generic Program Units):

فيما يلي، ثلاث نماذج تقليدية لتطبيق الوحدات البرمجية المولّدة:

- استخدام الوحدات البرمجية المولّدة، **كمكونات برمجية قابلة لإعادة الاستعمال.**
- استخدام الوحدات البرمجية، **كقالب لحالة الآلة.**
- استخدام الوحدات البرمجية، **للسيطرة على الرؤية.**

وفيمايلي، سنشرح كل تطبيق على حدة، وبالتفصيل.

الوحدات البرمجية المولّدة، كمكونات برمجية قابلة لإعادة الاستعمال :

مثلما رأينا في الحزمة البرمجية المولّدة Queues، فإن الوحدات البرمجية المولّدة، تسمح لنا بتوسيط (Parametrize) الوحدات غير المولّدة، حتى يمكن تطبيقها على مجال واسع من الإستخدامات.

والوحدات، هي مكوّنات قابلة للحل، تسمح للمطوّر ببناء نظم ضخمة ومعقدة، اعتباراً من وحدات برمجية مولّدة وموجودة. والفائدة الأولى من هذه الطريقة، هي أنّ يملك المطورون مستوى عالٍ من السرية في جودة حلولهم، إذ يبنونها من مكونات ذات سلوك محدد بشكل جيد.

مثال: لقد فحصنا منذ قليل، كيفية كتابة توصيف إجرائيــة الفـرز (Sort) ذات أهداف عامة، ولكن يمكننا إجراء الأحسن. والمثال الذي تمّ طرحه سـابقاً، يُطبّـق فقـط على عناصر من النوع المتقطع. فلنعتبر المثال التالي:

```
Generic
     type Item is private;
    type Index is (<>);
    type Items is array (Index) of Item;
    with function "<"(Left,Right:in Item) return Boolean is <>;
Procedure Sort(The_Items : in out Items);
```

فهنا، يمكن فرز شعاع من أي نـوع وأي طـول. ويتـم هـذا باسـتيراد النـوع Item كنوع خـاص، وبشـكل مضمـر (Implicitly)، يكـون تنفيـذ عمليـات الإسـناد، وفحـص المساواة أو عدمها، متاحاً. وبالإضافة لذلك، فإن استيراد أنـواع خاصـة، لا يمكـن أن يستورد، وبشكل مضمر، عمليـات علاقاتيـة. لذلـك، يجـب وبشـكل صريـح، اسـتيراد العامل "<"، بحيث يسمح لجسـم برنـامج Sort، بمقارنـة عنـاصر ذاتيـة مـن الشـعاع The_Items. وفيمايلي، استخدمنا خوارزميـة الفـرز السـريع (Quick-Sort)، مـن أجـل تنفيذ الإجرائية Sort، كما يلي:

```
procedure Sort(The_Items : in out Items) is
 Front : Index := The_Items'First;
 Back  : Index := The_Item'Last;
Procedure Swap is new Exchange(Item);
pragma Inline(Swap);
procedure Partition is
 Mid_Point : Constant Index := Index'Val;
    ((Index'Pos (Front) + Index'Pos (Back))/2);
 begin
   Outer :
    Loop
     While (The_Items(Front)<Mid_Value)
            and (Front /= The_Items'Last)
       Loop
         Front := Index'Succ(Front);
```

```
   End Loop;
  While (Mid_Value < The_Items (Back))
        and (Back /= The_Items'First)
   Loop
    Back := Index'Pred (Back);
   End Loop;
  if Front <= Back then
   if Front < Back then
    Swap(The_Items(Front), The_Items(Back));
   end if;
  if Front /= The_Items'Last then
   Front := Index'Succ(Front);
  end if;
  if Back /= The_Items'First then
   Back := Index'Pred(Back);
  end if;
  end if;
  exit Outer when (Front>Back)
            or ((Front=The_Items'Last)
            and (Back = The_Items'First));
 End Loop Outer;
end Partition;
Begin
 if The_Items'Length<2 then
  return;
 end if;
 Partition;
 if The_Item'First<Back then
  Sort(The_Items(The_Items'First..Back));
 end if;
 if Front < The_Items'Last then
  Sort(The_Items(Front..The_Items'Last));
 end if;
End Sort;
```

لاحظ هنا، استخدام واصفات الأنواع والأغـراض فـي جسـم الإجرائيـة، بحيـث استـخدمنا الواصـف Pred والواصـف Succ. ولاحـظ أيضـاً، كيفيـة حسـاب الغـرض

Mid_Point من الشعاع، منذ أن صرّحنا عن Index باستخدام الأنواع المتقطعة، وبالتالي، يمكن إعطاؤه قيماً مؤقتةً من أنواع مرقمة. وبسبب أنّ القسمة والجمع غير معرّفين على الأنواع المرقمة، فإننا نستخدم الواصف Pos، لتحويل قيم من Index إلى قيم صحيحة، والواصف Val، لتحويل القيم الصحيحة، لقيم من Index. وأخيراً، لاحظ أنّ القيمة Mid_Value، تحدد العنصر الأوسط، بدلاً من التصريح عن غرض جديد من Item.

ومعامل البرنامج الجزئي المولد "<"، اُستخدم في حلقات الـ While من الإجرائية Partition، بينما تم استيراد الإجرائية Partition إلى داخل الإجرائية Sort، وهو مرئي مباشرةً في جسم الإجرائية. ومع هذه الوحدة، يمكننا الآن تزويد نسخ مؤقتة تفرز محارف وفق الترتيب التصاعدي، كما يلي:

```
procedure String_Sort is new Sort(Item => Character,
                                  Index => Positive,
                                   Items => String,
                                     "<" => "<");
```

أما من أجل الفرز التنازلي لمجموعة محارف، فيجب تعديل ما سبق ذكره قليلاً، ليصبح كما يلي:

```
procedure String_Sort is new Sort(Item => Character,
                                  Index => Positive,
                                  Items => String,
                                    "<" => ">");
```

وفي أحد الفصول السابقة، قدمنا الحزمة البرمجية ranscendental_Functions، التي تصدر التوابع الفرعية Cos, Sin, Tan. ويمكن أن نجعل هذه الحزمة البرمجية مولّدة ببساطة، وذلك، باستخراج جميع المعلومات المتعلقة بالأنواع، مثلما فعلنا بالإجرائية Sort، كما يلي:

```
Generic
  type Real is digits <>;
package Transcendental_Functions is
  function Cos(Angle : in Real) return Real;
```

```
 function Sin (Angle : in Real) return Real;
 function Tan(Angle : in Real) return Real;
end Transcendental_Functions;
```

وبهذه الطريقة، نسخنا الحزمة البرمجية لتزويد التوابع الفرعية Sin, Cos, Tan لأيّ دقة مرغوبة. ومرةً ثانية، تعتبر واصفات النوع والغرض، كمفاتيح لكتابة جسم هذه الوحدة البرمجية، والتي تقبل أيّ نسخة مؤقتة فعلية.

استخدام الوحدات البرمجية، كقالبٍ لحالة الآلة:

لنعتبر الحالة المجرد لآلة Furnace، والتي عرضناها في فصل سابق. إن هذه الحزمة البرمجية، لا تعرف نوع معطيات الفرن Furnace، بينما (Rather) الحزمة البرمجية بأكملها، تمثل الفرن. وباستخدام حزمة برمجية مشابهة لهذه، فإن البرنامج يمكن أن يتضمن على الأكثر فرناً واحداً، بينما الحزم البرمجية لا يمكنها أن تنسخ قيماً من نوع معطيات.

وبتحويل الحزمة البرمجية كوحدة برمجية مولّدة، يمكننا إنتاج عدة أفران. ولاحظ أنّه تم جعل المولّد Furnace، يتطلب سطراً واحداً من التعديل على قسم توصيفه فقط، دون أيّ تعديل على جسمه، إذ يصبح قسم توصيفه كما يلي:

```
Generic package Furnace is
  ...
end Furnace;
```

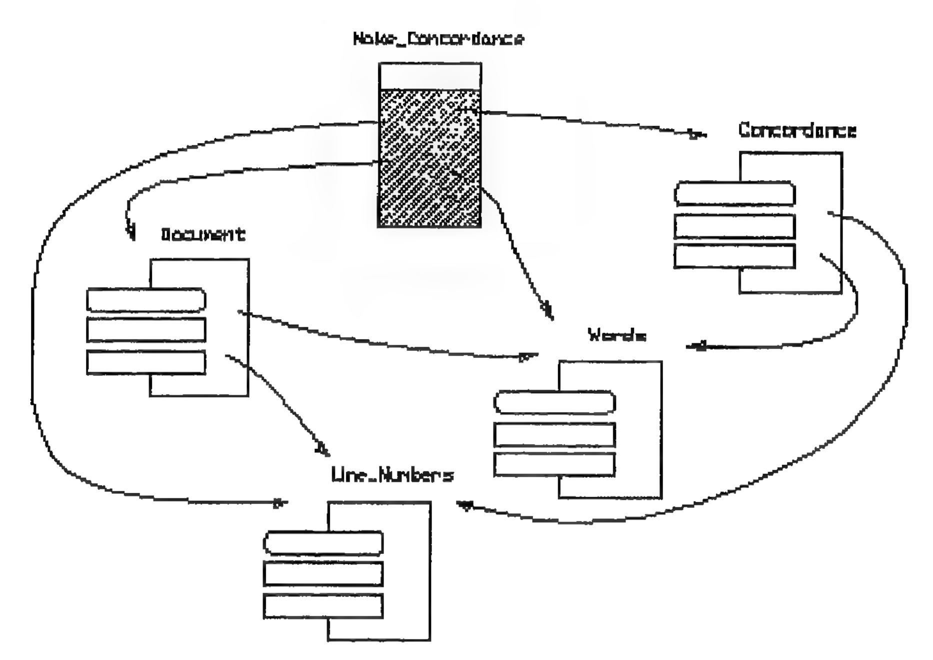
ولاحظ أنّ قسم التوليد فارغ (لم يتم التصريح عن المعاملات المولّدة). والهدف الوحيد من جعل الوحدة مولّدة، هو تبديل الحالة في جسمه. وفيما يلي، نسخ مؤقتة من الوحدة المولّدة:

```
package Car_Heater      is new Furnace;
package Garage_Heater  is new Furnace;
package Home_Furnace is new Furnace;
```

استخدام الوحدات البرمجية، للسيطرة على الرؤية:

تُستخدم الوحدات البرمجية المولّدة عادةً، كمكونات برمجية قابلة لإعادة الإستعمال. ولكن يمكن أن تستخدم لفصل (Decouple) أجزاء برنامج ضخم. فالفصل

يجعل كل مكون مستقلاً عن الآخر. فلنفرض أنّه توجد عدة حزم برمجية تعالج عدد وتكرار الكلمات الموجودة في نص ما، ولتكن هذه الحزم البرمجية مايلي: Words, Lines, Concordance .



الشكل ١٢ ـ ٢. Architecture of Make_Concordance.

فيمكن تعريف الحزمة البرمجية Concordance باستقلالية تامة عن تجريد الحزم البرمجية Words, Lines، باستخدام المعاملات المولّدة، من أجل استيراد المنابع التي تحتاجها الحزمة البرمجية Concordance. ووفق هذا، نستخدم المعاملات المولّدة، لتشير إلى المنابع بصراحة والتي يجب أن نجعلها مرئية للحزمة البرمجية. وبالتالي، بدلاً من أن نكتب الحزمة البرمجية Concordance كحزمة برمجية نظامية، فإنه يمكن التعبير عنها كمولّد، كما يلي:

```
Generic
  type Word is private;
  type Line is private;
  with function Value_Of (The_Word : in Word) return String;
```

```
  with function Image_Of (The_Line : in Line) return String;
package Concordance is
  procedure Start;
  procedure Add (The_Word : in Word; The_Line : in Line);
  procedure Make_Report;
  Overflow : exception;
end Concordance;
```

وللحصول على مرونة أكثر، استوردنا Word وline، كأنواع خاصة. وبالإضافة لذلك، يجب أن نستورد التابعين الفرعيين Value_Of وImage_Of، ليجعلا الحزمة البرمجية Concordance، قادرة على إنتاج تقرير.

ولا يمكن استخدام هذه الحزمة البرمجية مباشرةً، إذ يجب نسخها أولاً. فمثلاً، من أجل برنامج رئيسي (وليكن Make_Concordance)، يمكننا تضمين النسخ المتراكبة التالية:

```
package Local_Concordance is new Concordance
    (Word      => Words.word,
     Line      => Lines.Line,
     Value_Of => Words.Value_Of,
     Image_Of => Lines.Line'Image);
```

لاحظ هنا، أنّه يمكننا تطبيق واصف، كمعامل مولد فعلي يرتبط بـ Image_Of، لأنّ قواعد لغة ADA، تعالج الواصفات وكأنها وظائف خاصة. (وبما أننا فصلنا تصميم الحزم البرمجية Lines وwords عن الوحدة البرمجية Concordance، فيمكننا وبحرية تامة، إجراء تغيرات على الحزم البرمجية Words وlines، دون التأثير على الحزمة البرمجية Concordance، وأيضاً، يمكننا إعادة ترجمة النسخة المؤقتة).

مثال:

فيما يلي نستعرض برنامجاً متكاملاً، يتم من خلاله فرز عدة أشعة من مختلف الأنواع، وبأطوال مختلفة، بتطبيق برنامج جزئي مولد واحد (حاول أن تكتب هذا البرنامج، بأكثر من طريقة):

```
Generic
 type Element is Private;
 type List is array(Positive range <>) of Element;
 with function Compare_Two_Elements(E1,E2 : in Element)
   return Boolean is <>;
package Sort is
 procedure Exchange(E1,E2 : in out Element);
 procedure Bubble_Sort(L : in out List; List_Length: in Integer);
end Sort;
```

ويمثل هذا القسم، توصيف الحزمة البرمجية المولّدة، المسمّاة Sort. ومن خلالها، نستطيع استخدام إحدى الإجرائيتين Exchange وBubble_Sort، أو كلاهما معاً، في أية حزمة برمجية أخرى، نحتاج فيها تطبيق إحدى الإجرائيتين أو كلاهما. وبالتالي، يجب توليد نسخة مؤقتة عن الحزمة البرمجية المولّدة Sort، تلائم التطبيق المطروح. وفي هذه الحالة، فإن معاملات دخل الحزمة البرمجية المولّدة، تتمثل بالنوع الخاص Element، ومصفوفة العناصر الخاصة List ذات الطول غير المحدد، والوظيفة Compare_Two_Elements، الذي يجب تحديدها من قبل المبرمج، قبل توليد نسخة مؤقتة.

```
package body Sort is
 ---------- Exchange -------------
 procedure Exchange(E1,E2 : in out Element) is
  Temp : Constant Element :=E1;
 begin
  E1:=E2;
  E2:=Temp;
 end Exchange;
 --------- Bubble_Sort ----------
procedure Bubble_Sort(L:inout List;List_Length:in Integer) is
 Begin
  For I in 1..List_Length-1
  Loop
   For J in I+1..List_Length
```

```
   Loop
    if Compare_Two_Elements(L(J),L(I)) then
      Exchange(L(I),L(J));
    end if;
   End Loop;
  End Loop;
 End Bubble_Sort;
end Sort;
```

فوفق هذا الجزء، قد تمّ تحديد جسم الحزمة البرمجية المولّدة Sort، وخوارزميات الإجرائيتين Exchange وBubble_Sort .

```
WITH Text_IO; USE Text_IO;
WITH Sort;

PROCEDURE Generic_Example IS
package Int_IO is new Integer_IO(Integer);
package Flo_IO is new Float_IO(Float);
type Integer_List is array(positive range <>) of Integer;
type Float_List is array(positive range <>) of Float;
type String_List is array(positive range <>) of Character;
Int_List: Integer_List(1..5);
Flo_List : Float_List(1..4);
Str_List : String_List(1..8);
function I1_Is_Greater_Than_I2(I1,I2 : in Integer) return Boolean is
begin
 return I1>I2;
end I1_Is_Greater_Than_I2;
function F1_Is_Small_Than_F2(F1,F2 : in Float) return
Boolean is
begin
 return F1<F2;
end F1_Is_Small_Than_F2;
function S1_Is_Greater_Than_S2(S1,S2: in Character) return Boolean is
begin
```

```
 return S1>S2;
end S1_Is_Greater_Than_S2;
package Sort_Int is new Sort(Element => Integer,
                       List => Integer_List,
   Compare_Two_Elements =>  I1_Is_Greater_Than_I2);
package Sort_Flo is new
      Sort(Element => Float,
          List => Float_List,
   Compare_Two_Elements => F1_Is_Small_Than_F2);
package Sort_Str is new
      Sort(Element => Character,
          List => String_List,
      Compare_Two_Elements => S1_Is_Greater_Than_S2);
BEGIN
 put_Line("------ Enter 5 Integers -------");
 for I in 1..5 loop
  Int_IO.get(Int_List(I));
 end loop;
 Sort_Int.Bubble_Sort(Int_List,5);
 for I in 1..5 Loop
  Int_IO.put(Int_List(I));
 end Loop;
 new_Line;
 put_Line("------ Enter 4 Floats ------");
 for I in 1..4 loop
  Flo_IO.get(Flo_List(I));
 end Loop;
 Sort_Flo.Bubble_Sort(Flo_List,4);
 for I in 1..4 Loop
 Flo_IO.Put(Flo_List(I));
 end Loop;
 New_Line;
 put_Line("------ Enter 8 Characters -------");
 for I in 1..8 Loop
```

```
 get(Str_List(I));
 end loop;
 Sort_Str.Bubble_Sort(Str_List,8);
 for I in 1..8 Loop
   Put(Str_List(I));
 end Loop;
 New_Line;
END Generic_Example;
```

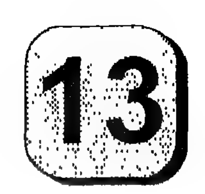
فوفق هذا الجزء، قد تمّ توليد الحزمة البرمجية Int_IO، من الحزمة البرمجية Text_IO.Integer_IO، والحزمة البرمجية Flo_IO، من الحزمة البرمجية Text_IO.Float_IO، لنتمكن من التعامل مع الأعداد الصحيحة، والأعداد الحقيقية، الممثلة بالفاصلة العائمة. كما تمّ تحديد الأنواع Integer_List, Float_List, String_List، كمصفوفات أحادية البعد من الأعداد الصحيحة والأعداد الحقيقية الممثلة بالفاصلة العائمة، ومحارف طول كل منها غير محدد. وبعد ذلك قد تمّ تحديد الأغراض Int_List, Flo_List, Str_List من الأنواع الآنفة الذكر، وبأطوال مختلفة، (٤، ٥، ٨). بعد ذلك، تمّ تعريف التوابع الفرعية I1_Is_Greater_Than_I2, F1_Is_Small_Than_F2, S1_Is_Greater_Than_S2، لتعيد قيمةً منطقيةً تمثل ناتج مقارنة (أكبر أو أصغر) غرضين، لهما إحدى الأنواع Integer, Float, Character بالترتيب. وبعد ذلك، تمّ توليد ثلاث نسخ مؤقتة من الحزمة البرمجية Sort، تتلاءم مع ثلاثة أنوع من المعطيات، يتم من خلالها الفرز تصاعدياً أو تنازلياً (حسب تعريف توابع المقارنة الفرعية السابقة الذكر). هذه النسخ، هي ما يلي:

- Sort_Int، من أجل فرز مصفوفة أعداد صحيحة أحادية البعد.
- Sort_Flo، من أجل فرز مصفوفة أعداد حقيقية ممثلة بالفاصلة العائمة أحادية البعد.
- Sort_Str، من أجل فرز مصفوفة محارف أحادية البعد.

بعد ذلك، تمّ استخدام هذه النسخ مع الحزم البرمجية Int_IO, Flo_IO, Text_IO، من أجل قراءة وفرز (تصاعدياً أو تنازلياً) وإخراج ناتج الفرز على الشاشة، لمصفوفة من الأعداد الصحيحة أحادية البعد (عدد عناصرها ٥)، ولمصفوفة من الأعداد الحقيقية، ممثلة بالفاصلة العائمة أحادية البعد (عدد عناصرها ٤)، ولمصفوفة من المحارف أحادية البعد (عدد عناصرها ٨).







مسألة التصميم الثالثة:
حزمة برمجية لشجرة مولّدة
Generic Tree Package

تعريف المسألة

تحديد الأغراض

تحديد العمليات

تأسيس الرؤية

تأسيس واجهة التخاطب

تقييم الأغراض

زرع كل غرض

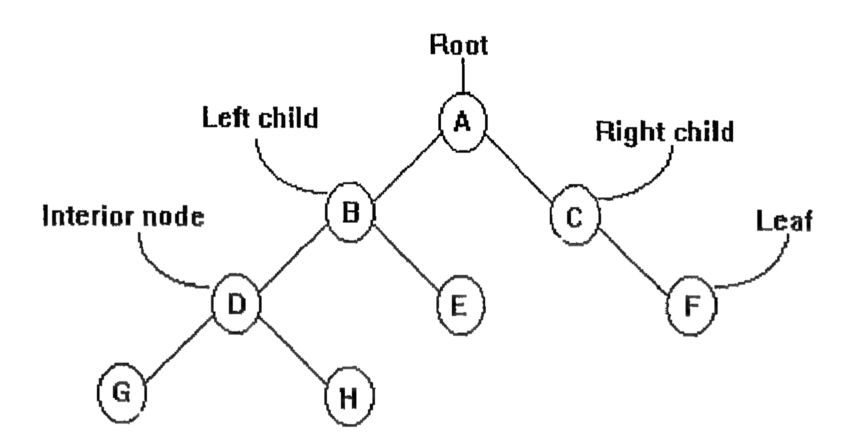


تعتبر الحزم البرمجية، ووحدات البرامج المولّدة، أدوات فعالة تساعد المبرمج بإدارة تعقيد الحلول البرمجية. ومثلما رأينا في الفصل ١٢، تسهل الوحدات المولّدة إعادة استخدام المركبات البرمجية. وهكذا مكونات، توسع مقدرة لغة ADA بتحزيم تجريدات جديدة، في شكل يمكن تطبيقه بسهولة في مجالات مسائل مختلفة.

وفي هذا الفصل، سنطبق طريقتنا غرضية التوجه، لبناء مركب يمكن إعادة استخدامه، وهو الشجرة الثنائية. ولا تمثل الشجرة نوعاً أساسياً في ADA، ولذلك، يجب أن نبني نوع معطياتنا المجرد الخاص بنا. وأكثر من ذلك، نريد تطبيق هذا التجريد على أشجار من أنواع مختلفة؛ وفي الواقع، نريد أن نستخدم من جديد تجريدنا لأشجار صحيحة، أشجار تسجيلات، ومن الممكن حتى أشجار، قيمها تمثل أغراضاً لنوع معطيات مجرد آخر. وعندما ننتهي من بناء هذه المركبات، سنرى كيف يمكننا تطبيق وحدتنا لشجرة مولّدة، بإكمال جسم الحزمة البرمجية Concordance من الفصل ٥.

١٣ ـ ١ ـ تعريف المسألة (Define the Problem) :

يستخدم المبرمجون الأشجار في العديد من التطبيقات المختلفة، مثل، البحث في قواعد معطيات، والتمثيلات الوسيطة للبرامج الحاسوبية، والتخطيط، وترجمة اللغات الطبيعية. ونجد في العالم الحقيقي، العديد من الأغراض التي تشبه الأشجار بسلوكها: تنظيم خطاب، ومخطط علم الوراثة، تمثل أمثلةً قليلة. وتأخذ الأشجار بشكل أساسي فعاليتها، من قدرتها على تمثيل علاقة هرمية بين العناصر. وهذه الوظيفة، هي التي تميزها عن التجريدات، مثل الأرتال، والتي تكون جميعها وحيدة البعد.



الشكل ١٣ ـ ١. شجرة ثنائية.

مثلما نرى في الشكل ١٣ ـ ١، يمكن تمثيل الشجرة كمجموعة من العقد المرتبة بطريقة هرمية. ويمكن أن تملك كل عقدة قيمة؛ في هذا الشكل، نرى ثمانية عقد مسماة من A إلى H. وقد صُممت العقدة A كجذر للشجرة. (تُرسم الأشجار المجردة، بوجود جذرها في الأعلى، بشكل مختلف عن نظيرها العضوي) ويربط المسار البسيط كل عقدة مع العقدة المجاورة، في المستوى التالي. وبالتالي، نقول بأنّ B, C أولاداً للعقدة A؛ وA أصل العقدتين B, C، وأنّ B, C أخوة. ففي شجرة ثنائية، يمكن لكل عقدة أن تحتوي على ولدين على الأكثر، سميا ولد يمين أو يسار. فإذا لم تحتو العقدة على أي ولد (مثل E)، فإنها تكمن على حدود الشجرة، وتُدعى بورقة أو عقدة طرفية. وفي غير ذلك، تكون العقدة داخلية.

ويتمثل عملنا بتطوير وحدة مولّدة، تحقق هذا التجريد للشجرة الثنائية. وأكثر من ذلك، سنضيف المتطلبات التي تحتاجها الوحدة التي صممناها، ليتم تطبيقها على أشجار، قيم عقدها هي من أي نوع.

١٣ ـ ٢ ـ تحديد الأغراض (Identify the Objects) :

لقد عرفنا المسألة بشكل كاف؛ وتحتوي المسألة على غرض واحد ذي اهتمام أساسي. وسنسمي هذا الصف من الأغراض بالنوع Tree. وبالتأكيد، يمكننا تعريف ا لـ Tree بتصريح بسيط للنوع، مثل ذلك الذي عرفناه في الفصل ٦. ولكن لا يمكن ضبط تجريدنا، بتصدير نوع غير معلب بكل بساطة. ولذلك، نحتاج بالأحرى، لإخفاء تمثيل النوع Tree في حزمة برمجية، ومن ثمّ نصدر فقط العمليات التي تدعم تجريدنا لشجرة.

ولاحظ بأنّ الشجرة تحتوي أيضاً غرضاً ذا أهمية ثانوية. ولقد ذكرنا سابقاً، بأنّه يمكن لكل عقدة أن تحتوي قيمة؛ ولذلك، يجب أن نهتمّ بصف الأغراض، حيث نأخذ هذه القيم. والآن، لنفترض بأنّ قيمة العقدة يمكن أن تكون من أي نوع غير محدد. فمثلما سنرى فيما بعد، سيلعب هذا القرار دوراً هاماً بتعريف الرؤية الخارجية لمركبنا. وفي الواقع، هذا هو القرار الذي يسمح لنا بتعميم تجريد شجرتنا، وعمل وحدة مولّدةٍ منها.

وقد حذفنا أيضاً، إمكانية استخدام مركب شجرتنا لقيم عقد، ستكون من النوع الخاص المحدود. ولأسبابنا أساس متين، فالأنواع المحدودة الخاصة، تمنع عمليات الإسناد وفحص المساواة. وإن غياب هذه العمليات، يجعل من الصعب إسناد قيم لعقدة من شجرة، ومن ثم إيجادها من جديد. وعلى أية حال، فالأنواع غير المحدودة (التي تتضمن بقية صفوف الأنواع الأخرى)، غير مقيدة بهذه الطريقة؛ ويمكننا تطبيق شجرتنا على أوسع صفوف أنواع، مع وضع قيود أصغرية فقط، على نوع قيمة العقد التي سندعوها Item.

١٣. ٣ تحديد العمليات (Identify the Operations) :

تعتبر هذه الخطوة حرجةً في بناء تجريدنا. ففي الحقيقة، إن تعيين العمليات، يؤسس سلوك تجريدنا مثلما يُرى من الخارج. ومثلما هو الحال مع بقية التجريدات الأخرى التي فحصناها، يمكننا تصنيف هذه العمليات، كمختارات وبنّاءات.

وتتضمن بنّاءات الشجرة، جميع العمليات التي تغير حالة غرض. وبشكل خاص، نحتاج لعملية تنسخ شجرة لأُخرى، ولعملية ثانية، تفرغ محتوى شجرة:

- Copy -- Copy one tree object to another.
- Clear -- Make the tree empty.

ونحتاج أيضاً، لعملية تضيف عقدةً جديدةً لشجرة محددة. وتتمثل الصعوبة هنا، بتعيين هذه العملية بطريقة لا نفقد بها أثر أي عقدة موجودة. وتتمثل طريقتنا، بخلق عقدة جديدة، وجعلها جذر شجرة أُخرى. وسنربط الجذر القديم كولد للعقدة الجديدة:

- Construct -- Create a new node with the given value and make it
  -- the new root; take the old root and attach it as a child
  -- of the new root.

ولإعطاء أعظم درجة من المرونة، سندع زبون هذه العملية يختار لأي ولد سنربط الجذر القديم. وتعطينا Construct طريقةً لتغيير شكل الشجرة بإضافة عقدة جديدة. ولكن نحتاج أيضاً، لعملية تسمح لنا بإعادة تنظيم العلاقات بين العقد. وبالتالي، سنُضمن عملية تبدل أحد أولاد عقدة الجذر، مع شجرة أُخرى:

- Swap_Child -- Swap the named child of the root node with
  -- another tree.

ومثل Construct، فقد تمّ تعريف دلالات هذه العملية بطريقة لا نستطيع وفقها فقد أثر أيّة عقدة بسهولة. وبالتالي، عندما نبدّل ولد عقدة الجذر مع الشجرة، بدلاً من فقدان ولد هذه الشجرة الجزئية ببساطة، نحفظه بربطه كجذر للشجرة الثانية.

ونحتاج لبنّاء آخر، حيث أنه لدينا عقدة ذات قيمة، فإننا نحتاج لطريقةٍ ما، لتغيير تلك القيمة. وعندها، سنضمن العملية التالية:

- Set_Item -- -- Set the Value of a node to the given value.

ولنتجه إلى مختارات هذا التجريد، ستتمثل طريقتنا بتوفير عمليات متكاملة، توازي تجمع البناءات. وبالتالي، بما أننا نضمن بنّاء ينسخ شجرةً لأُخرى، سنعرّف أيضاً مختاراً يفحص فيما إذا كان للشجرتين قيماً متساوية. وبشكل مشابه، تُفرغ Clear شجرة. وكنتيجة لذلك، سنعرف مختارات تفحص فيما إذا كانت شجرة فارغةً حقاً:

• Is_Equal -- Return True if the Two given trees have an equal value.

• Is_Null -- Return True if the given tree is empty.

وموازاةً للبناء Swap_Child، فانه يجب أن نضمن عمليةً تعيد ولداً من عقدة جذر شجرة محددة. وبما أنّ ولد عقدة أيضاً شجرة، فإن هذه العملية تعيد في الواقع، غرضاً من النوع Tree:

• Child_Of -- Return a child of the root node of the given tree.
-- the name of the child (left or right) is specified
-- by the client.

وأخيراً، نحتاج لعملية توازي البنّاء Set_Item:

• Item_Of -- Return the value of the root of given tree.

وكما في جميع أمثلة التصميم السابقة، يجب أن نعتبر الآن الشروط الإستثنائية. وبشكل خاص، يجب أن نعتبر إمكانية التنفيذ خارج الذاكرة، خلال عمليات مثل Copy وConstruct. وبالإضافة لذلك، تتطلب بعض العمليات استخدام شجرة غير فارغة. وبشكل خاص، Set_Item, Swap_Child, Item_Of, Child_Of فإن جميع هذه العمليات، تتطلب ألا تكون الشجرة المستخدمة فارغةً. وعلى سبيل المثال، من الواضح أنه لا يمكن معالجة Set_Item، إذا لم توجد عقدة جذر.

ويمكننا تقسيم هذه الشروط الإستثنائية إلى صفين:

• Overflow لم تبق ذاكرة كافية لإتمام العملية المرجوة.

• Tree_Is_Null لا يمكن تنفيذ العملية المرجوة، لأنّ الشجرة المحددة فارغة.

وهكذا يكتمل توصيفنا للنوع Tree، ولكن أيضاً، يجب أن نعتبر النوع Item. ومن أجل ذلك، يجب أن نعكس نظرتنا للنوع Tree. وبدلاً من الإنشغال بالعمليات التي يجب أن نصدرها، يجب أن نعتبر العمليات التي يجب استيرادها Item لحزمتنا البرمجية للشجرة المولّدة. ومثلما ناقشنا، نريد جعل هذه المكونات عامةً قدر الإمكان. وهكذا، فقد عرفنا النوع Item، كأي نوع غير محدود. وبالتالي، على الأقل،

يجب أن نسمح بالإسناد، وفحص المساواة على أغراض من هذا النوع. وهذا ليس صعباً كما يبدو، مثلما سنرى في مقطع لاحق. وتعطينا معاملات ADA المولّدة، الآلية الدقيقة للتعبير عن هذه المضامين.

١٣ ـ ٤ ـ تأسيس الرؤية (Establish the Visibility):

في هذه المسألة، فإن تأسيس الرؤية لأغراض مهمة، يكون مباشراً. وسنصدر فقط النوع Tree وعملياته من هذه الحزمة البرمجية المولّدة، ولكن لا تستطيع الحزمة البرمجية نفسها رؤية أي شيء، باستثناء النوع Item. وبهذه الطريقه، سنبني مركباً ذا ارتباط ضعيف جداً.

وهذه الحالة شائعة في الوحدات المولّدة. فمثلما درسنا في الفصل السابق، ليست الوحدات المولّدة مفيدة فقط في بناء مركبات برمجية، ولكن تستخدم أيضاً، للتحكم بالرؤية بين وحدات الترجمة. ومثلما نرى هنا، فبتعريف تجريد شجرتنا بشكل مستقل عن أي تجريد آخر (باستثناء النوع Item)، نكون بالفعل، قد فككنا ارتباطه عن أي وحدة أُخرى في فضاء حلّنا.

١٣. ٥ تأسيس واجهة التخاطب (Establish the Interface) :

في هذه النقطة، يمكننا التقاط قرارات تصميمنا الخاصة بتجريد الشجرة إلى وحدة مولدة:

```
type Item is private;
package trees is
type tree is private;
type Child is (Left, Right);
Null_Tree : constant Tree;
procedure Copy (From_The_Tree : in Tree;
                        To_The_Tree : in out Tree);
procedure Clear (The_Tree : in out Tree);
procedure Construct (The_Item : in Item;
                        And_The_Tree : in out Tree;
On_The_Child : in Child);
procedure Set_Item (Of_The_Tree : in out Tree;
```

```
                    To_The_Item : in Item);
procedure Swap_Child (The_Child : in Child;
                    Of_The_Tree : in out Tree;
And_The_Tree : in out Tree);
function Is_Equal (Left : in Tree;
                    Right : in Tree) return Boolean;
function Is_Null (The_Tree : in Tree) return Boolean;
function Item_Of (The_Tree : in Tree) return Item;
function Child_Of (The_Tree : in Tree;
                    The_Child : in Child) return Tree;
   Overflow, Tree_Is_Null : exception;
Private
...
end trees;
```

ومثلما حققنا في العديد من أنواع المعطيات المجردة الأُخرى، كتبنا البنّاءات كإنتاج راثيات، وتمّ تصدير المختارات كوظائف، وتمّ التصريح عن الإستثناءات صراحة. ولاحظ أيضاً، كيف استوردنا النوع Item : باستخدام نوع خاص، أكدنا أنّه بإمكاننا نسخ هذا المركب مع أي نوع غير محدد. ومثلما درسنا في الفصل السابق، يوفر هذا التصريح ضمنياً عمليات الإسناد وفحص المساواة، كما تتطلب مسألتنا.

١٣ ـ ٦ ـ تقييم الأغراض (Evaluate the Objects) :

لقد ناقشنا في الفصل ١٠ ثلاث طرق لتقييم الأغراض. وقد شملت هذه الطرق، الإقتران، والاختيار، وتصنيف العمليات. وفي هذا الفصل، قدمنا طريقتين تشملان تصنيف العمليات :

١ ـ **تحديد الشروط الإستثنائية**، فمعظم العمليات، ستحتوي على حالات خاصة، من أجلها لا يمكن استخدامها. ويجب اعتبار هذه الحالات، والإستثناءات المعرفة لها.

٢ ـ **يجب أن تقترن الشروط الإستثنائية، بمختارات موافقة**: فبعض الحالات التي تمّت معالجتها كالإستثناءات، يجب أن تمتلك عمليةً مقترنةً يمكن استخدامها لمراقبة الحالة.

تعين الشروط الإستثنائية (Identify Exceptional Conditions) :

تمثل الوثوقية إحدى الأهداف الأساسية لهندسة البرمجيات. ومثلما ناقشنا في الفصل ٢، فالبرنامج الموثوق، يمنع الإخفاق. وتتمثل الخطوة الأولى في المنع، بتعيين الشروط التي يمكن أن تسبب الإخفاق. وأكثر من ذلك، أيضاً كما في الفصل ٢، يجب أن تُبنى الوثوقية منذ البدء – منذ التصميم. ومن غير الممكن بناء تنفيذ برمجي موثوق، لواجهة تخاطب غير موثوقة. ويجب تعيين الشروط التي يمكن أن تسبب إخفاقاً" في مستوى واجهة التخاطب.

العديد من شروط الإخفاق، أو الشروط الإستثنائية التي يمكن تعيينها ببساطة، بتقييد القيم التي ستعمل عليها العملية. ويمكن تحقيق ذلك، باستخدام أنواع. فلنعتبر وظيفة الجذر التربيعي:

function Square_Root (Radicand : Float) return Float;

فسيخفق هذا التابع بإعادة قيمة معبرة عند تطبيقه على أعداد سالبة. ويمكننا صراحةً صياغة هذا الشرط، باستخدام نوع جزئي:

function Non_Negative is Float range 0.0..Float'Last;
function Square_Root(Radicand : Non_Negative)return Non_Negative;

فعندما عرّفنا بشكل جيد، القيم التي سيُطبق عليها Square_Root، نكون قد عرّفنا أيضاً بشكل جيد، القيم التي سيُعيدها.

وإن بعض الشروط الإستثنائية، من الصعب تحديدها. وبهذه الحالات، يمكن أن تصرح واجهة التخاطب عن استثناء يحدد الشرط. ولا توجد طريقة من أجل وحدة Trees، لتؤكد بأنّ Item_Of لن يتم استدعاؤه أبداً مع شجرة فارغة. حتى أن التابع Item_Of، يجب أن يعمل بشكل موثوق ومتنبأ عنه مسبقاً، في هكذا استدعاء. وتمثل الإستثناءات آلية ADA للسماح لـ Item_Of، ليكون موثوقاً دون إعادة قيمة.

ويجب أن تقترن الشروط الإستثنائية، بمختارات موافقة:

فإذا كنا قد شرحنا للمستخدم متى يبرز الإستثناء Tree_Is_Null، فمن المحتوم أن نذكر الشرط الذي تكون فيه الشجرة فارغة. ولقد جعلنا هذا الشرط صريحاً، في

تجريدنا بتعريف التابع المنطقي Is_Null. وبتعريف الإستثناء والشرط الموافـق، نجعـل التجريد أكثر اكتمالاً، وموثقاً ذاتياً، ونوفـر أيضـاً للمسـتخدمين، مرونـة فحـص الشـرط بأنفسهم، فضلاً عن انتظار بروز الإستثناء. فلنفـرض أنّـه لدينـا غـرض مـن النـوع Tree يُدعى بـ T، وأردنا فحـص كـل عنصـر فـي الفـرع اليسـاري منـه. فيمكـن كتابـة هـذه الخوارزمية، باستخدام إما الإستثناء أو الشرط، كما يلي:

```
While not Is_Null loop
... Item_Of(T)...
    T := Child_Of(T,Left);
end loop;
```

```
  begin
loop
   ... Item_Of(T)...
    T := Child_Of(T,Left);
  end loop;
exception
  when Tree_Is_Null => null;
end;
```

ويعتبر أسلوب ترميز المثال الأيسر سيئاً، بسـبب أنّـه اسـتخدم بشـكل أساسـي الإستثناء كتعليم goto الضمنية. وإن الوصول إلى نهاية الشجرة ليس استثناءً ولا خطـأً؛ ونتوقع حدوثه بكل لحظةٍ ننفذ بها الترميز. وأما مثال الطرف الأيمن، فصريح ومفهوم.

وفي الواقع، فقد لاحظنا بأننا صدرنا شرط "is null" مرتين: مـرةً كوظيفـة ومـرة مثل T := Null_Tree؛ وبشكل عام، إن تصدير أيّة عملية أكثر مـن مـرة، يعتـبر عمـلاً سيئاً، نظراً لتقليص قابلية صيانة التضافر.

١٣ - ٧ - زرع كل غرض (Implement Each Object) :

دعنا ننتقل لوجهة النظر الداخلية لهذا المركب ولنعتبر تنفيذه البرمجي(زرعه) تاماً. فأولاً، يجب أن نقرر تنفيذاً برمجياً للنوع Tree، الذي سيشتق منـه شـكل بقيـة تنفيذنا. والطريقة الأكثر وضوحاً، تتمثل بالتنفيذ البرمجي للنوع Tree بطريقـةٍ واقعيـة، تعكس تجريده في فضاء المسألة. وبالتالي، يمكننا تمثيـل Tree كمؤشـرٍ لعقـدة، بينمـا النوع Node فيمثل تسجيلةً تحتوي على عنصر وشجرتين جزئيتين. ويتضمن هذا التنفيذ البرمجي أنواعاً تراجعية، ولهذا يجب أن نطبق نوعـاً غـير تـام، مثلمـا تتطلـب قواعـد

ADA. وعلى أية حال، يمكننا الإستفادة من هذه المتطلبات لإخفاء التعريف التام، للنوع غير التام. وبالتالي، في القسم الخاص من هذه الحزمة البرمجية، يمكننا كتابة:

```
type Node;
type Tree is access Node;
Null Tree : constant Tree := null;
```

وفي جسم المركب، يمكننا إكمال تعريف النوع Node، كما يلي:

```
type Node;
Record
   The_Item      : Item;
   Left_Subtree,
   Right_Subtree : Tree;
end record;
```

ودعنا الآن نفحص التنفيذ البرمجي لكل عملية، وفق ترتيب ظهورها في جسم الحزمة البرمجية. ومثلما سنرى، فإننا نحتاج فقط، لتطبيق بعض الخوارزميات البسيطة. وإن بعضاً من هذه الخوارزميات رائعة جداً، بسبب طبيعة الشجرة التراجعية. وعلى أية حال، يجب أن لا تُخدع بهذه البساطة. ففي الواقع، يمكن أن يتعجب القارئ الحريص، لماذا تعذبنا ببناء الشجرة كنوع خاص. ويوجد سبب جيد لتعليب قرارات تصميمنا. فبتحديد الرؤية الخارجية لتجريدنا، نوفر للزبائن التفاصيل المزعجة لقرارات تصميمنا. وتعطينا هذه الطريقة، كمبرمجين، حرية اختيار أي تمثيل نرغبه، ما دامت تحقق دلالات الرؤية الخارجية. ويمكننا تغيير تفاصيل التنفيذ البرمجي هذا، دون تشويش الزبون (أيضاً، يمكننا إجبارهم على إعادة الترجمة).

وتمثل Copy، العملية الأولى التي سندرسها. وبسبب تراجعية الشجرة (هذا يعني، تعرّف الشجرة الثنائية كعقدةٍ جذر، مع شجرتين جزئيتين كأولاد)، يمكننا إجراء التنفيذ البرمجي لـ Copy، بشكل تراجعي. ويجب على خوارزميتنا أولاً، معرفة فيما إذا كانت الشجرة فارغة. فإذا كانت لذلك، نجعل ببساطة، الشجرة الهدف فارغةً. وفي غير ذلك، فإننا نحجز عقدةً جديدةً كجذر للشجرة الهدف، ونستدعي Copy مرة ثانية، لنسخ أولاً، الشجرة الجزئية اليسرى، ومن ثمّ الشجرة الجزئية اليمنى:

```
Procedure Copy (From_The_Tree  : in Tree;
           To_The_Tree     : in out Tree) is
Begin
  if From_The_Tree = null then
   To_The_Tree := null;
  Else
   To_The_Tree :=
    new Node'(the_Item       => From_The_Tree.The_Item,
         Left_Subtree  => null,
         Right_Subtree => null);
  Copy(From_The_Tree.Left_Subtree,
 To_The_Tree.Left_Subtree);
  Copy(From_The_Tree.Right_Subtree,
To_The_Tree.Right_Subtree);
  End if;
exception
  when Storage_Error => raise Overflow;
end Copy;
```

لاحظ كيف يمكننا استخدام تكتل لبناء قيمة عقدة، قد تمّ خلقها من جديد في تعليمة واحدة فقط. وبما أنّه يمكن لهذا المجمع إبراز الإستثناء Storage_Error، فإنه يجب أن نُضمن معالج استثناء ليلتقط هذا الإستثناء، ومن ثمّ يبرز استثناءً ذا إسم معبر (Overflow) .

وهذا جسم Clear بسيط: فهو يطبق اصطلاحنا، بأنّ شجرة فارغة تمثل شجرة قيمتها null:

```
 procedure Clear (The_Tree : in out Tree) is
 Begin
    The_Tree := null;
 end Clear;
```

ويبدأ Construct بشكل مشابه لبداية Copy ، بحجز عقدة جديدة. وعلى أية حال، فبما أنّ الشجرة المحددة يمكن أن تحتوي بعض العقد، فإن دلالات هذه العملية، تتطلب بأنّ نحفظ العقد الموجودة، بوصلها على شكل شجرة جزئية لعقدة جديدة. وبالتالي، تتضمن الإجرائية معاملاً، يحدد الولد الذي يجب استخدامه:

```
procedure Construct (The_Item : in Items;
                     And_The_Tree : in out Tree;
                     On_The_Child : in Child ) is

Begin
   if On_The_Child = Left then
      And_The_Tree := new Node'(The_Item => The_Item,
                                Left_Subtree  => And_The_Tree,
                                Right_Subtree => null);
   Else
      And_The_Tree := new Node'(The_Item => The_Item,
                                Left_Subtree  => null,
                                Right_Subtree  => And_The_Tree);
   end if;
Exception
   when Storage_Error => raise Overflow;
end Construct;
```

ومثلما أجرينا في Copy، لاحظ كيف يجب أن نقدم معالج استثناء، ليبحث في شرط ممكن لـ Overflow.

ويتطلب Set_Item فقط تعليمةً واحدة. ويجب ببساطة، إعطاء القيمة The_Item من جذر العقدة. وفي حال Of_The_Tree تكون فارغةً، تبرز هذه التعليمة Constraint_Error، التي نعالجها بإبراز الإستثناء Tree_Is_Null :

```
procedure Set_Item (Of_The_Tree : in out Tree;
                    To_The_Item : in Item) is
Begin
   Of_The_Tree.The_Item := To_The_Item;
Exception
   when Constraint_Error => raise Tree_Is_Null;
end Set_Item;
```

ويمثل Swap_Child، آخر بنّاء يجب تنفيذه برمجياً. وبشكل مشابه لـ Construct، تتضمن العملية معاملاً يميزً الولد من شجرة المنبع، الذي يجب أن يحل محله شجرة الهدف. ولاحظ بأنّه يجب أن نقدم غرضاً مؤقتاً، نستخدمه لحفظ

الشجرة الجزئية الولد، عندما نبدل موضع الولد مع شجرة الهدف. وبدلاً من استبعاد هذه الشجرة، نحفظها كشجرة هدف :

```
procedure Swap_Child (The_Child : in Child;
                      Of_The_Tree : in out Tree;
                      nd_The_Tree : in out Tree ) is
   Temporary_Node : Tree;
 Begin
        if The_Child = Left then
           Temporary_Node := Of_The_Tree.Left_Subtree;
           Of_The_Tree.Left_Sbtree := And_The_Tree;
        Else
           Temporary_Node := Of_The_Tree.Right_Subtree;
           Of_The_Tree.Right_Sbtree := And_The_Tree;
        end if;
        And_The_Tree := Temporary_Node;
     Exception
        when Constraint_Error => raise Tree_Is_Null;
  end Swap_Child;
```

ولنتجه الآن إلى مختارات الأشجار. فلقد وجدنا بأنّ Is_Equal، تحتوي على خوارزمية تراجعية بسيطة، تتبع نموذجاً وجدناه في البنّاء Copy. وتتحقق خوارزميتنا، من أنّ جذور الأشجار المُعطاة تتطابق. فإذا كانت هذه هي الحالة، فإننا نستدعي بشكل تراجعي Is_Equal، لفحص مساواة الشجرتين الجزئيتين:

```
procedure Is_Equal (Left : in Tree;

Begin
   if Left.The_Item /= Right.The_Item then return False;
   Else
      return Is_Equal(Left.Left_Subtree,Right.Left_Subtree)
      and then Is_Equal(Left.Right_Subtree, Right.Right_Subtree);
   end if;
Exception
```

```
    when Constraint_Error => return False;
end Swap_Child;
```

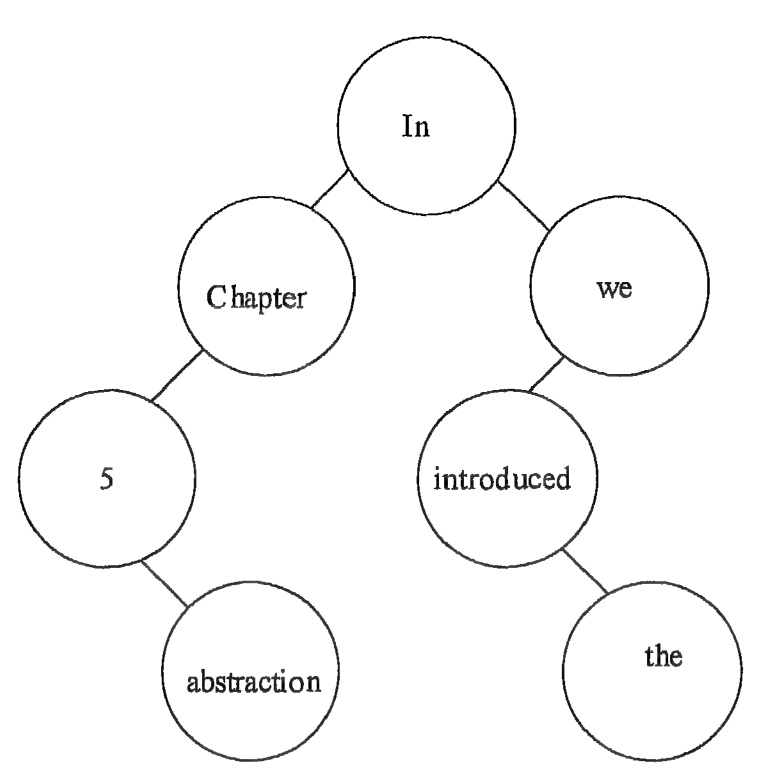
إن أجسام بقية المختارات بسيطة. ويفحص Is_Null المساواة مع null ، وItem_Of وChild_Of، يخفيان الوصول المباشر لمركبات التسجيلة Node. ومن أجل المختارين الأخيرين، يجب أن نحمي أنفسنا من Constraint_Error ممكن، والذي يمكن أن يبرز في حال كانت الشجرة المحددة فارغة. ونُضمن معالج استثناء، الذي يبرز Tree_Is_Null في مكانه:

```
function Is_Null (The_Tree : in Tree) return Boolean is
    Begin
        return The_Tree = null;
    end Is_Null;

    function Item_Of (The_Tree : in Tree) return Item is
    Begin
        return The_Tree.The_Item;
     Exception
         when Constraint_Error => return Tree_Is_Null;
    end Item_Of;

    function Child_Of (The_Tree : in Tree;
                                The_Child : in Child) return Tree is
    Begin
        if The_Child = Left then
            return The_Tree.Left_Subtree;
        Else
            return The_Tree.Right_Subtree;
        end if;
     Exception
         when Constraint_Error => return Tree_Is_Null;
  end Item_Of;
```

وبهذا يكتمل تنفيذنا البرمجي للحزمة البرمجية للشجرة. وقبل أن نترك هذا الفصل، على أية حال، دعنا نعتبر تطبيقاً معبراً.



الشكل ١٣ ـ ٢. The_Concordance.

في الفصل ٥، قدمنا تجريد الفهرسة الأبجدية كغرض. ويقدم برنامجنا الكلمات، وأرقام أسطرها الواحدة بعد الأُخرى، إلى هذه الآلة ذات الحالة-المجردة، ويجب أن تحفظ الفهرسة الأبجدية هذه الكلمات، وأرقام الأسطر الموافقة بطريقة ما. ومثلما نرى في الشكل ٢، إحدى الإستراتيجيات الممكنة، تتمثل باستخدام الشجرة. وبالتالي، كلما راكمنا كلمات، نقوم بإضافتها إلى الشجرة وفق الترتيب الأبجدي. ولأول وهلة، لا يبدو هذا الشكل، آخذاً الكلمات وفق الترتيب الأبجدي. ولكن لاحظ، ومن أجل كل عقدة، بأنّ قيمة شجرتها الجزئية اليسرى، دائماً أصغر من قيمة شجرتها الجزئية اليمنى. ويُبين الشكل حالة الفهرسة الأبجدية، بعد أن سجلنا الكلمات الأولية لهذا المقطع. ثم لنعتبر كيف أضفنا الكلمة التالية، of. وانطلاقاً من جذر الشجرة، نسأل فيما إذا كانت الكلمة الحالية (of) أصغر، أو تساوي، أو أكبر من قيمة العقدة. ففي هذه الحالة، إن قيمتها أكبر، لذلك، ننتقل للولد الأيمن. ونسأل نفس

السؤال، ثم ننتقل إلى الولد الأيسر (بما أنّ السلسلة المحرفية of، أصغر من السلسلة we). وفي العقدة التالية المدعوة بـ introduced ، نختار الإنتقال مرة ثانية لليمين، لأنّ of أكبر. وبالنظر إلى the نجد أنّ of تكون أصغر. وعلى أية حال، ففي هذه المرة، لا يوجد أولاد. وبما أننا على حدود الشجرة، ولم نجد ارتباطاً، نُنشئ عقدةً جديدة، ونربطها كولدٍ أيسر للعقدة المسماة بـ the.

ونعكس المعالجة، لخلق ترتيب أبجدي للكلمات. وانطلاقاً من عقدة الجذر، ننتقل لليسار. فإذا كان هنالك ولد يساري، نستدعي هذه الخوارزمية بشكل تراجعي. وإذا لم يوجد، نطبع قيمة العقدة، ثمّ نستدعي الخوارزمية بشكل تراجعي على الولد الأيمن.

وهذا العبور شائع في شجرات البحث، ويُدعى بالعبور المرتب. وتتمثل الفائدة من هذه الطريقة، بأنها تسمح بحشر كلمات جديدة. وعندما ننتهي من بناء الفهرسة الأبجدية، فإننا لسنا بحاجة لفرز جميع الكلمات؛ ونحتاج فقط، لننتقل على الشجرة لخلق الترتيب الأبجدي.

ودعنا نتجه لجسم الحزمة البرمجية Concordance. ففي الواقع، إن قيمة كل عقدة في الشجرة، يجب أن تكون أعقد بشكل طفيف عما وصفناه. وبما أنّه يمكن ظهور كلمة واحدة أكثر من مرة في وثيقة، فإنه يجب أن نملك وسيلةً لحفظ جميع مراجع الكلمة. ولحسن الحظ، لدينا مركب سيؤدي العمل. ففي الفصل السابق، بنينا نسخةً مولّدةً للحزمة البرمجية Queues. وبتطبيق نفس المفهوم، يمكننا معالجة مراجع الكلمات، كرتلٍ من أرقام أسطر. وبالتالي، كلما وجدنا كلمة، نضيف رقم السطر للرتل الموافق لهذه الكلمة. وعندما نكون جاهزين لعرض الفهرسة الأبجدية، نستخرج ببساطة أرقام الأسطر من واجهة الرتل، الذي يعيد لنا هذه الأرقام، وفق الترتيب الذي تمّ إدخال هذه الأعداد فيه إلى الرتل.

وإن العقدة في شجرة الفهرسة الأبجدية، هي تلك التسجيلة المؤلفة من كلمة، ورتل يحتوي أرقام الأسطر. ويمكننا التعبير عن قرارات تصميمنا حول نافذة العمل لجسم الحزمة البرمجية Concordance، كما يلي:

```
with Text_IO, Queues, Trees;
package body Concordance is
package Line_Number_Queue is new Queues(Line_Numbers.Number);
  type Reference is access Line_Number_Queue.Queue(100);
  type Node is
     Record
        The_Word : Words.Word;
        The_References : Reference;
     end record;
  package Word_Tree is new Trees(Node);
  The_Concordance : Word_Tree.Tree;
  procedure Start is ...
  procedure Add (The_Word : in Words.Word;
                 The_Number : in Line_Numbers.Number) is ...
  procedure Make_Report is ...
end Concordance;
```

لاحظ كيف يجب أن نقيد التصريح عن الأغراض ذات النوع Queue ، لأن النوع يتطلب مميزاً. وقرار جعل ١٠٠ مرجعاً كحد أعلى لكل كلمة، هو اختياري.

وبالأخذ بعين الإعتبار، التنفيذ البرمجي لكل من هذه الإجرائيا ت، نبدأ برؤية كيف يمكن استخدام الموارد المتوفرة، من قبل مركباتها البرمجية القابلة لإعادة الاستخدام. فعلى سبيل المثال، يعطي Start قيمةً بدائيةً ثابتةً، لحالة هذه الحزمة البرمجية. وفي هذه الحالة، الحالة الوحيدة المرتبطة مع جسم هذه الحزمة البرمجية، تتمثل بالغرض The_Concordance . وبالتالي، يمكن بناء Start، على قمة بناء الشجرة Clear :

```
procedure Start is
Begin
   Word_Tree.Clear(The_Concordance);
end Start;
```

ونمرر إلى Add، كلمة ورقم السطر. وفي جسمه، نستدعي ببساطة إجرائيةً أخرى، التي سنصرح عنها محلياً لجسم الحزمة البرمجية:

```
procedure Add (The_Word : in Words.Word;
               The Number : in Line_Numbers.Number) is
Begin
  Insert(The_Word,      the_Number,      In_The_Tree      =>
  The_Concordance);
Exception
  when Word_Tree.Overflow => raise Overflow;
End Add;
```

وسيتضح سبب إضافة هذا المستوى غير المباشر، فيما بعد. وقبل أن ندرس جسم الإجرائية المحلية Insert، دعنا نضيف تابعاً فرعياً محلياً آخراً سنحتاجه فيما بعد. وتذكر بأنّ العمليات العلاقاتية من أجل String، تقارن غرضين من String عنصراً. وعلى أية حال، يمكن أن نلتقي بكلماتٍ بحالات مختلفة، كما بنينا الفهرسة الأبجدية. (إن قواعد ADA من أجل العمليات العلاقاتية، تعرّف الكلمتين ADA وADA ككلمتين غير متساويتين، على سبيل المثال.) وبالتالي، يجب أن نضيف عملية، تقلب جميع محارف السلسلة المحرفية إلى محارف سفلية:

```
function Lower_Case (The_String : in String) return String is
   Result : String(1..The_String'Length) := The_String;
   Offset : constant := Character'Pos('a') - Character'Pos('A');
Begin
   for Index in Result'Range
      Loop
        if Result(Index) in 'A'..'Z' then
          Result(Index)   :=   Character'Val(Character'Pos(Result(Index))   +
          Offset);
        end if;
      end loop;
   return Result;
end Lower_Case;
```

ويمكن أن يتبع الآن Insert، الخوارزمية التراجعية التي وصفناها باكراً. فأولاً، دعنا نفحص Insert بتفصيل أكثر. فمن أجل شجرةٍ محددة، نرى فيما إذا كانت فارغة. وإذا كانت كذلك، فإننا نستدعي Construct، لبناء عقدةٍ جديدة. وفي غير ذلك، نفحص فيما

إذا كانت الكلمة المحددة أصغر، أو تساوي، أو أكبر من كلمة العقدة الحالية. فإذا ساوتها، فإننا ببساطة، نضيف رقم السطر المحدد، إلى رتل أرقام الأسطر، باستدعاء بنّاء الرتل Add. وإذا كانت الكلمة أصغر من كلمة العقدة الحالية، ننتقل إلى الولد الأيسر، وفي غير ذلك، ننتقل إلى الولد الأيمن. وفي كلتا الحالتين، نحفظ مؤشراً للعقدة الحالية، ومن ثمّ نستدعي Insert بشكل تراجعي. وعندما نعود من Insert، نعلم بأنّ الشجرة المحددة (Temporary_Tree) تمّ تغييرها، لذلك، نعيد ربطها إلى آخر ولدٍ زرناه:

```
procInsert (The_Word : in Words.Word;
            The_Line : in Line_Numbers.Number;
            In_The_Tree : in out Word_Tree.Tree) is
Temporary_Node : Node;
   Trmporary_Tree : Word_Tree.Tree;
Begin
   if Word_Tree.Is_Null(In_The_Tree) then
      Temporary_Node :=
            Node'(The_Word => The_Word,
                 The_References => new
                 Line_Number_Queue.Queue(100));
      Line_Number_Queue.Add(The_Line,
      Temporary_Node.The_References.all);
      Word_Tree.Construct(Temporary_Node,
                        And_The_Tree => In_The_Tree,
                        On_The_Child => Word_Tree.Left);
   Else
      Temporary_Tree := In_The_Tree;
      Temporary_Node := Word_Tree.Item_Of(Temporary_Tree);
      if Lower_Case(Words.Value_Of(The_Word)) =
         Lower_Case(Words.Value_Of(Temporary_Node.The_Word))
          Then
           Line_Number_Queue.Add(The_Line,
           Temporary_Node.The_References.all);
      elsif Lower_Case(Words.Value_Of(The_Word))<
        Lower_Case(Words.Value_Of(Temporary_Node.The_Word)) then
           Temporary_Tree := Word_Tree.Child_Of(In_The_Tree,
           Word_Tree.Left);
           Insert(The_Word, The_Line, Temporary_Tree);
```

```
            Word_Tree.Swap_Child(Word_Tree.Left,
                                 Of_The_Tree => In_The_Tree,
                                 And_The_Tree => Temporary_Tree);
        Else
            Temporary_Tree := Word_Tree.Child_Of(In_The_Tree,
            Word_Tree.Right);
            Insert(The_Word, The_Line, Temporary_Tree);
            Word_Tree.Swap_Child(Word_Tree.Right,
                                 Of_The_Tree => In_The_Tree,
                                 And_The_Tree => Temporary_Tree);
        endif;
    end if;
end Insert;
```

لاحظ كيف استثمرنا نفوذ موارد مركب شجرتنا. وفي غياب هذا المركب، سيكون تنفيذنا البرمجي أكثر تعقيداً، لأن المطوّر يريد أن يضاعف الكثير من عملنا السابق.

ولنتوجه إلى التنفيذ البرمجي لـ Make_Report، باستخدام خوارزمية تراجعية :

```
Procedure Make_Report is
   procedure Display (The_Tree : in Word_Tree.Tree) is ...

   procedure Traverse (The_Tree:in Word_Tree.Tree) is ...
Begin
   Text_IO.New_Line;
   Traverse(The_Concordance);
end Make_Report;
```

وتمثل Display إجرائيةً محلية، حيث من أجل عقدة محددة، تعرض قيمة الكلمة وجميع أرقام الأسطر الموافقة لها. وهنا، يمكننا تطبيق بعض التسهيلات من Text_IO، لتقديم الشكل الذي نرغبه؛ وسنناقش هذه التسهيلات أكثر، في الفصل ١٨. لقد عقدّنا خوارزميتنا بشكل طفيف، بتقديم الغرض Previous_Line. وبدلاً من تكرار المراجع التي تظهر على نفس السطر، لا نطبع رقم سطر، إلا إذا كانت قيمته مختلفة عن آخر قيمةٍ طبعناها :

```
procedure Display is
   Temporary_Node : Node;
```

```
    The_Line : Line_Numbers.Number;
    Previous_Line : Line_Numbers.Number := Line_Numbers.Number'Last;
    function "=" (X, Y : Line_Numbers.Number) return Boolean
        renames Line_Numbers."=";
Begin
    Temporary_Node := Word_Tree.Item_Of(The_Tree);
    Text_IO.Put(Words.Value_Of(Temporary_Node.The_Word));
    Text_IO.Set_Col(20);
    While Line_Number_Queue.Length_Of
        (Temporary_Node.the_References.all)>0
    Loop
        Line_Number_Queue.Remove(The_Line,
        Temporary_Node.the_References.all);
        If The_Line /= Previous_Line then
            Text_IO.Put(Line_Numbers.Number'Image(The_Line));
            Previous_Line := The_Line;
        end if;
    end loop
    Text_IO.New_Line;
end Display;
```

وأخيراً نفحص جسم Travers. وتنفذ هذه الإجرائية برمجياً، العبور المرتب الذي وصفناه سابقاً:

```
procedure Traverse (The_Tree : in Word_Tree.Tree) is
Begin
    if not Word_Tree.Is_Null(The_Tree) then
        Traverse(Word_Tree.Child_Of(The_Tree, Word_Tree.Left));
        Display(The_Tree);
        Traverse(Word_Tree.Child_Of(The_Tree, Word_Tree.Right));
    end if;
end Traverse;
```

وهكذا ينتهي زرع جسم Concordance.







المهــام
Tasks

شكل المهام بلغة ADA

تعليمات المهام

تطبيقات مهام لغة ADA



في مجال مسألة العالم الحقيقي، تحدث عادةً عدة نشاطات بنفس الوقت. فعلى سبيل المثال، يمكن لطيار آلي أن يراقب، وبشـكل مستمر حساسـات سـرعة الريـاح، وزاوية الهجوم، وينتظر طلبـات المسـتخدم، ويسيطر علـى عـدة أجهـزة مسـتقلة عـن بعضها، مثل الجنيحات، وغير ذلـك.و بالإضافـة لذلـك، فـإن بعـض الأجهـزة، مثـل مساعدات الملاحة، يمكن أن تستخدم مقاطعات ذات بنية صلبة غير متزامنة، كطلبات خدمة.

فإذا أردنا تطوير حل برمجي لمسائل من هذا النـوع، سـنجد بـأنّ معظـم لغـات البرمجة عالية المستوى، تقدم قليلاً، أو لا تقدم المساعدة للتعبير عن هكـذا نشـاطات موازية. وبالتـالي، فإنـه يجـب اللجـوء إلـى الخدمـات التـي يقدمهـا نظـام الإسـتثمار المضيف، أو يجب كتابة عدة إجرائيات خاصة بلغة المجمع، لإدارة المهام.

وبشكل عام، لا يوجد أي حل مقبول. فعندما نخـرج عـن نطـاق اللغـات عاليـة المستوى، فإن المنتج البرمجي، لن يبقى قابلاً للنقل، كما أنّه تصعب صيانته، بسـبب انفصاله لعدة قطع. وبالإضافة لذلك، فإن كتابـة منتـج برمجـي متعـدد المهـام، يعتـبر نشاطاً صعباً. وعند البرمجة بلغة «التجميع» المجمع، يكون مـن الصعـب، إن لـم يكـن من المستحيل، التعبير وبصراحة عن العلاقات المؤقتة أو التحكم بشكل موثـوق بـردود فعل عدة مهام مختلفة.

وبلغة ADA، إن المهمة ببسـاطة، تمثـل كيانـاً يعمـل علـى التـوازي مـع عـدة وحدات برمجية أخرى. ومنطقيـاً، يمكننـا اعتمـاد مفهـوم حـل مكـون مـن عـدة مهـام مسـتقلة. وفيزيائيـاً، يمكـن تنفيـذ المهـام علـى عـدة نظـم حاسـوبية، أو نظـم متعـددة المعالجـات، أو إجـراء التنفيـذ المتداخـل علـى معالج وحيـد. ومهمـا كـان التمثيـل الفيزيائي، فإن التجريد لحل يتضمن عدة مهام، يعتبر شيئاً طبيعيـاً، ويؤخـذ مباشـرةً من تجريدنا لفضاء المسألة. وما نرغبه نحن، يتجلى بالقدرة على تمثيل هذه النشاطات الحقيقية المتوازية، بجميع مستويات حلّنا. واليوم، نحن مرتاحون تماماً بالتعـامل مـع الأعداد الحقيقية، دون أن ننشغل بمستوى البت (Bit).

بما أنّ المهام مستقلة عن بعضها البعض كلياً، فإنه يجب أن توجد بعض الوسائط للتعبير عن والإتصال فيما بينها. وبشكل أمثلي، نود أن يكون هذا الإتصال صريحاً وموثوقاً.

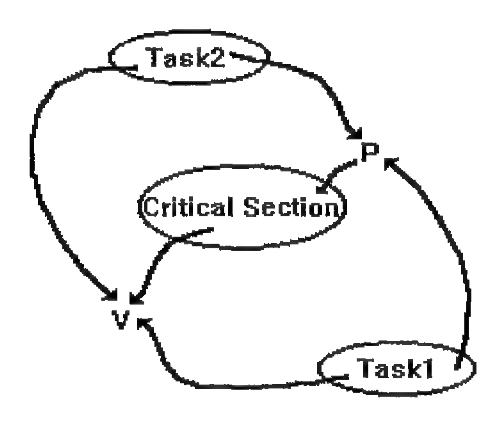
فعلى سبيل المثال، لنعتبر نظاماً ذا مهمتين، المهمة الأولى، تأخذ عينات من لوحة المفاتيح، وتجمعها على شكل أسطر نصية (لتكن هذه المهمة Producer)، والمهمة الثانية، تأخذ الأحرف المحفوظة بأسطر، وترسلها عبر معدل/كاشف (مودم Modem) (لتكن هذه المهمة Consumer). فمن البديهي، أنه لا يمكننا التنبؤ عن زمن تفاعل المهمتين مع بعضهما البعض. ويمكننا الإنتظار لعدة دقائق، من أجل دخل للوحة المفاتيح، أو يمكننا فجأة الحصول على فاصل لدخل المعطيات، لـ 80 كلمة بالدقيقة. وبالإضافة لذلك، يجب أن ننهي بنجاح، الحوار الصحيح مع المودم، وربما أن نحذر إعادة الإرسال في حال الخطأ.

وتوجد طريقتان رئيسيتان، تعبران عن الإتصال بين هاتين المهمتين. فالطريقة الأولى، مشابهة لتوجيه رسالة من خلال صندوق بريد. والمهمة Producer، تبني مجموعةً من مدخلات لوحة المفاتيح، ومن ثم، عند الإنتهاء من تجميع سطر طبيعي، يوضع ما تمّ تجميعه في منطقة ذاكرة مشتركة ومعروفة بالنسبة للمهمتين (صندوق بريد). وبعدها، يتم تحديد بعض أنواع المؤشرات، للتنويه بأنّ صندوق البريد يحتوي رسالة. وبالتالي، فالمهمة Consumer، تجمع الرسائل المتواجدة في صندوق البريد.

وحسب قواعد هذا النوع من الإتصال، فقط، يمكن لمهمةٍ واحدة أن تدخل صندوق البريد، في وقت واحد (المنع المتبادل Mutual Exclusion)، وإذا حاولت كلتا المهمتين الدخول بنفس الوقت، فيجب على إحداهما الإنتظار، كي لا تتداخل مع الأخرى. وبالإضافة لذلك، إذا لم يوجد أي دخل، فإنّ المهمة Consumer تنتظر في صندوق البريد. ومن جهة أخرى، إذا كانت المهمة Consumer مشغولة في معالجة رسالة، فإن المهمة Producer، ببساطة، تتابع توجيه الرسائل وإيداعها في صندوق البريد، وتعود إلى نشاطها من جديد.

وبمفهوم لغات البرمجة، يمكننا زرع طريق إتصالٍ مـا، باستخدام مـا يدعـى بــ Semaphores أو Monitors.

وحسب الشكل١٤ ـ ١، يمكن اعتبار صنـدوق الـبريد، كمنبـع، محمـي بمقطـع حرج من الترميز، بحيث يمكن لمهمة واحدة فقط، أن تدخله في لحظـة واحـدة. وفـي هذه الحالة، فإن المؤشر هو Semaphore، ويتم إبـرازه بواسـطة عمليـة مؤشـرة بــ V، ويتم خفضه بواسطة عملية مؤشرة بـ P. ومن أجل الإتصال البسيط بيـن المهـام ، فـإن الطريقة مقبولة بشكل جيد. وعلى أيـة حـال، فـإن هـذه الطريقـة، لا تعـالج الحـالات المعقدة بشكل جيد.



الشكل ١٤ ـ ١. الإتصال بين المهام بواسطةٍ semaphore.

وعند الحاجة لأكثر من مهمة من النوع Producer، ولأكثر مـن مهمـة مـن النـوع Consumer، بدلاً من مهمتين فقط، فمن الضروري الحصول على نظـام أفضليـات، مـن أجل حجز المنابع. وعلـى الأكـثر، فـإن هـذا النـوع مـن إتصـال المهـام غـير مـتزامن. وبالنتيجة، فإنه من الصعب التعبير عن حل يُجيب عن تساؤلات زمنيـة معينـة، مثـل "معلومات لوحة المفاتيح سترسل عبر مودم بأقل من ٥٠٠ ميلـي ثانيـة، بعـد اسـتقبال سطر مكتمل." ، بينما لا يمكن لمهمة واحدة بالفعل، معرفة وجود بقيـة المهـام (كـل مهمة ترى فقـط الــ Semaphore، ولا تـرى بقيـة المهـام). ومـن الصعـب إن لـم يكـن

مستحيلاً، أن تكتشف مهمةٌ، خطأ مهمة أخرى. ويمكن التصور بأن المهمة Consumer يمكن أن تتعطل، وأن المهمة Producer تتابع خلق أسطر معطيات الدخل، التي ستفقد بسبب عطل المهمة Consumer.

وإن الطريقة الأكثر طبيعيةً لمعالجة تفاعل المهام، هي معالجة كل مهمة كإجراء تسلسلي متصل، كما هو موضح في الشكل ١٤ ـ ٢.



الشكل ١٤ ـ ٢ المهام، كإجراء تسلسلي متصل.

فبدلاً من أن تكون غير متزامنة، فإن بعض المهام، تكون متزامنة بالزمن والمكان عندما تكون متصلة، بطريقة مشابهة لشخصين يتحادثان فيما بينهما. وباستخدام مثالنا، فبعد خلق سطر دخل من قبل المهمة Producer ، فإنها تستدعي مدخلاً من المهمة Consumer، وهذا يشير بأنّ المهمة جاهزة للإتصال. وفي اللحظة التي تقبل فيها المهمة Consumer المدخل، تمر الرسالة (في اتجاه واحد، أو في اتجاهين)، وبعدها تعمل المهمتان بشكل منفصل عن بعضهما البعض، حتى تصبح المهمة Producer جاهزةً لتسليم رسالة أخرى. وأي تزامن صريح يعرف بموعد (Rendezvous). وفي هذا النموذج، إذا كانت إحدى المهام جاهزةً للدخول أو القبول، قبل أن تكون المهمة الثانية في نقطة الموعد، فإن لتلك المهمة ثلاث خيارات. وهذه الخيارات مايلي: إما الإنتظار بشكل غير محدد، أو الإنتظار لفترة زمنية معينة، أو يمكنها الدخول/ القبول مهمة أخرى جاهزة للإتصال. والفائدة من هذه الطريقة، تتمثل بأنّ الإتصال بهذه الطريقة هو أكثر وثوقيةً. فإذا تعطلت إحدى المهام أو تأخرت، يمكن في هذه الحالة لمهمة أخرى أن تكتشف ذلك، وتأخذ قياسات الأفضلية. وبالإضافة لذلك، إذا كان إتصال المهام متزامناً، يمكننا وببساطة أكثر، التعبير عن العلاقات المتزامنة بين مهمتين.

وتحتوي لغة ADA بنىً، تسمح لنـا بـالتعبير مباشرةً عـن بنيـة حلـول متعـددة المهام، باستخدام الإجـراءات المتصلـة تسلسلياً. وبعكس معظم بقيـة اللغـات عاليـة المستوى، فليس من الضروري الرجوع إلى لغة مضيفة لتنفيذ (لزرع) المهمة.

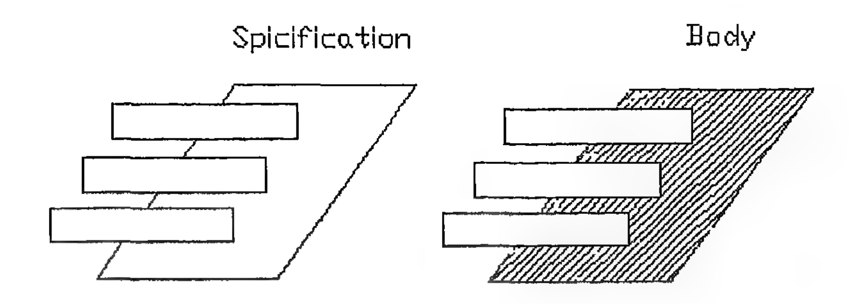
١٤ ـ ١ ـ شكل المهام بلغة ADA (The Form of ADA Tasks):

تعتبر المهمة إحـدى ثـلاث وحـدات برمجيـة رئيسـية بلغـة ADA (الوحدتـان الباقيتان هما البرامج الجزئية، والحزم البرمجية). ولنعتبر وحدة برنامج رئيسـي لتكـون وبشكل مضمر مهمة، ولكن يمكن للمـبرمج أن يصـرّح عـن مهـام أخـرى صراحـةً، فـي وحدات برمجية أخرى. ويمكن للمبرمج أن يضع المهام في قسم التصريح من أي وحدة برمجية، مثل الكتل، وجسم برنامج جزئي، وجسم مهمة، أو حزمة برمجيـة مكتبيـة. ولا يمكن لمهمة مستقلة أن تـترجم بشـكل منفصـل، ولكـن يمكـن تعليبهـا فـي حزمـة برمجية مكتبية، للحصول على نفس التأثير. ولا يمكن لمهمة أن تتواجد منعزلة، فهـي ترتبط بأصلها (Parent)، والذي يمثل بشكل أساسي، الوحدة التي يتم بها التصريح عن المهمة.

وبلغة ADA، لسـنا بحاجـة لبـدء (Initiate) المهمـة صراحـةً. وبـالعكس، فإنـه تنشط المهام بعد بناء (Elaborated) جسمها، في نهاية قسـم تصريح الأصـل. وإذا تـمّ التصريح عن عدة مهام في نفس قسم التصريح، فسيتم تنشـيط جميـع هـذه المهـام فـي نهاية قسم التصريح، إذ أنّ ترتيب التنشيط غير محدد.

ومن جهـة أخرى، لا تنتهي مهمـةٌ أصـل، حتـى تنتهـي جميـع المهـام الأولاد (Children). وتنتهي المهمة بشـكل طبيعـي، عندمـا يصـل تنفيذهـا إلـى نهايـة جسـم المهمة، وإنتهاء جميع المهام المرتبطة بها، إن وجدت.

ومثل الحزم البرمجيـة، تتكـون المهمـة مـن قسـمين شـكل ١٤. ٣، وهمـا قسـم توصيف المهمة، وقسم جسم المهمة. ويمثل قسم توصيف المهمة، الواجهة بين المهمة وبقية الوحدات البرمجية، بينما يتكون جسم البرنامج، من قسم المهمة القابل للتنفيـذ. ويمكن لقسم توصيف المهمة، ولقسم جسمها، أن ينفصلا نصياً.



الشكل ١٣ - ٣. رموز المهام في لغة ADA.

توصيف المهمة (Task Specifications) :

يقدم قسم توصيف المهمة إسم غرض المهمة (أو نوع المهمة، مثلما سنناقشه فيما بعد)، والمداخل المرئية بالنسبة لمستخدم المهمة. وقسم التوصيف، يعرّف مسارات الإتصال (المداخل) الصالحة لبقية المهام. على سبيل المثال، من أجل المهمة Consumer المعرّفة سابقاً، يمكن لقسم التوصيف أن يكون على الشكل التالي:

```
task Consumer is
    entry Recive_Message(A_Message : in String);
End Consumer;
```

إن توصيف مدخل، له شكل مشابه لتوصيف برنامج جزئي، إذ يتألف من إسم المدخل، متبوعاً بمعاملات صورية، لتصف نوع المعاملات(الرسائل)، والتي تمّ تمريرها خلال الموعد. ويمكن لهذه المعاملات الصورية أن تأخذ أحد النماذج In, Out, In Out، بشكل مشابه للبرامج الجزئية. وهذه النماذج، تحدد جهة التدفق لكل رسالة، بالنسبة للمهمة التي تتضمن التصريح عن المدخل.

ويمكن استدعاء مدخل مهمة، من أي نقطة مسموح منها استدعاء برنامج جزئي، بدءاً من برنامج جزئي (بما في ذلك البرنامج الرئيسي)، أو مهمة، أو جسم حزمة برمجية ، أو كتلة، بالرغم من أنه لا يوجد معنى لأن تستدعي مهمة، مداخلها الخاصة. فإذا حاولت مهمة أن تتصل مع نفسها، يمكن خلق شرط موت (Deadlock)،

لأن الموعد مستحيل. ويحدث الموت عندما تنتظر المهـام منبعـاً، لا يمكـن مطلقـاً أن يصبح حراً.

ولنعتبر توصيف المهام التالية، الأكثر تعقيداً :

```
task Protected_Stack is
    entry Pop (Element : Out Integer);
    entry Push(Element : In Integer);
End Protected_Stack;
```

وهناك عدة صفات لهذا التصريح، تستحق الإشارة إليها. أولاً، يمكـن لتوصيـف مهمة أن يحتوي فقط، تصريح مهام (و توصيفات تمثيلات)، بعكس الحزم البرمجيـة، التي تصدر عدة أنواع مختلفة من الكيانات. ومثلما هو الحال بالحزم البرمجية، يجـب على مستخدم المهمة الرجوع إلى مداخل معينة، وذلك باستخدام الترمـيز المسمّى. ولا يمكن تطبيق العبارة use على مداخل المهام، لذلك، من الضروري دائماً إسباق المدخل المستدعى بإسم المهمة. والمثال التالي، يوضح ذلك:

```
Protected_Stack.Pop(My_Value);
Protected_Stack.Push(36);
```

ولا يختلف استدعاء مدخل عن شكل تنشيط برنامج جزئـي. وبالحقيقـة، يمكـن إعادة تسمية المداخل مثل الإجرائيات. والمثال على ذلك ما يلي:

```
procedure  Protected_Pop (Element : Out Integer) renames
Protected_Stack.Pop;
```

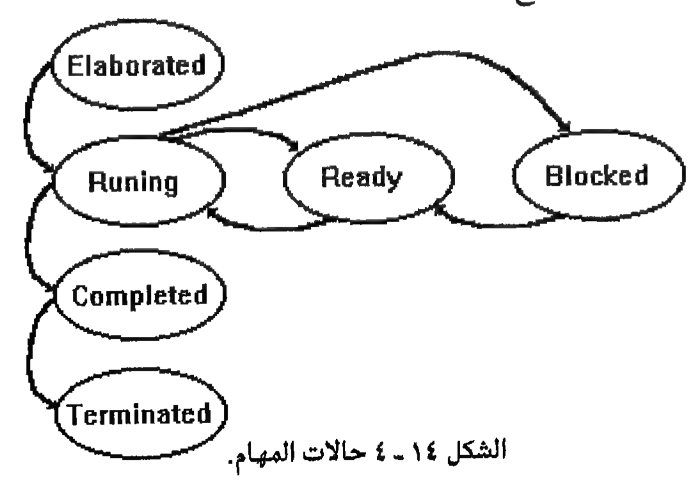
وعندئـذ نسـتطيع وببسـاطة، إحضـار Protected_Pop. بينمـا لا يمكننـا تطبيـق العبارة use على مهمة، فإن إعادة التسمية (renames)، هي غالباً مفيدة جداً، وبشـكل خاص، عندما يكون إسم المهمة وإسماء مداخلها طويلة، ونرغب بتقصيرها، وإعطائهـا إسماً ذا دلالة لعمل المدخل.

وأيضـاً، إن اسـتدعاء المدخـل، شـبيهٍ لاسـتدعاء برنـامج جزئـي، بينمـا دلالـة التحضير تختلـف بشـكل كـامل. فـإذا اسـتطاعت عـدة مهـام اسـتدعاء نفـس البرنـامج الجزئي، عندئذ يمكن لعدة مهام فعلياً، تنفيذ نفس البرنامج الجزئي في وقـت واحـد،

وعندها نقول، بأنّ ترميز البرنامج الجزئي مشترك (أو متعدد الدخول (Reentrant). وعلى أية حال، يوجد رتل ضمني (Implicit queue)، مرتبط بتصريح كل مدخل. فإذا استدعت عدة مهام نفس المدخل، فقط مهمة واحدة (ابتداء من المهمة التي استدعت المدخل أولاً) فإنه يسمح لها بتنفيذ الموعد (Rendezvous). وستنتظر جميع المهام الأخرى في الرتل الضمني (Implicit queue)، وفق ترتيب الوصول لكل مهمة (First_in First_out)، وليس حسب درجة الأفضلية للمهمة. وفي حال استدعاء مهمتين لنفس المدخل في لحظة واحدة، فسيتم إختيار المهمة إختيارياً. ومثلما سنلاحظ فيما بعد، يمكن لمهمة ترك الرتل قبل إكمال الموعد. ويمكن حدوث هذا في حال وجود الشرط (Timeout) (بعد مضي القسم الأعظم من زمن الإنتظار)، ويمكن لمهمة نشطة، أن تتواجد في إحدى الحالات الخمس التالية:

- Running (تنفيذ): المعالج يخدم المهمة في الوقت الحالي.
- Ready (جاهزية): المهمة منفكة (ليست في حالة تأخير، وليست منتظرة موعداً) ومنتظرة من أجل المعالجة.
- Blocked (إنتظار): المهمة في حالة تأخير، أو منتظرة موعداً.
- Completed (إكتمال): المهمة أنهت تنفيذ سلسلة تعليماتها.
- Terminated (إنتهاء): المهمة غير نشطة أبداً، أو لم تعد نشطة أبداً.

والشكل ١٤ ـ ٤ يوضح مخطط التفاعل بين هذه الحالات.



الشكل ١٤ ـ ٤ حالات المهام.

هنا، نلاحظ بأنّه، مثل أي تصريح آخر، فـي البـدء يجـب إنجـاز المهمـة، ولا ينشط إلا في نهاية إنجاز منطقة التصريـح المحصـورة. وخـلال حياتهـا النشـطة، فـإن حالـة المهمـة تـأخذ إحـدى الحـالات Running, Ready, Blocked. وإن الحالـة Running، تشير بأنّ المهمة تمتلك منابع المعالجة؛ والحالة Ready، تشير بأنّ المهمة تنتظر منابع المعالجة، لكنها جـاهزة للتنفيـذ. والحالـة Blocked، تعنـي بـأنّ المهمـة تنتظر حدثاً، مثل موعد. ومثلما سنناقش بتفصيل أكثر فيمـا بعـد، فـإن دلالـة ADA لا تتطلب بأن يحتاج ترتيب المهام لخوارزمية إلى تقسيم الزمن. وبالنتيجة، عندما تكـون المهمة نشطة، يمكنها متابعة التنفيـذ حتـى تجهـز مهمـة ذات أفضليـة أعلـى. وعنـد إنتهاء مهمة من تنفيذ تعليماتها، تنتقل لحالـة الإكتمـال (Completed)، وتنتظـر حتـى تكتمل جميع المهام المرتبطة بها. وبعد اكتمال المهمـة، لا تهتـم المهمـة بـأي موعـد. وفي حال إنتهاء جميع المهام الأبناء من النشاط، ستنتهي المهمة. ويرتبط بكل مهمـة، بما في ذلك البرنامج الرئيسي، أفضليـات ثابتـة، تشـير إلـى درجـة الأهميـة. ويمكـن للمستخدم صراحةً تعين درجة الأفضليـة باسـتخدام الــ Pragma Priority. وتـأخذ الــ Pragma قيمةً صحيحة مـن النـوع Priority (مـن الحزمـة البرمجيـة System)، حيـث يتحدد مجال القيم عند تنفيذ (زرع) البرنامج. إذ أنّ أعلى قيمة، تشير إلى أعظم درجة أهمية. وإذا لم تعين الأفضلية صراحة لمهمة محددة، فسيسند المترجم قيمـة. ولا تؤثـر الأفضلية على الترتيب، الذي وفقه تنتظر المهام في رتل من أجـل التخديـم، والسـبب يعود إلى أنّ التخديم في لغة ADA يتبع المبدأ First_in First_out .

ويتمثـل تـأثير الأفضليـة بالمساعدة فـي تخصيـص منـابع المعالجـة، مثـل المعالجات، أو أماكن الذاكرة، إلى مهام متوازية، فـي حـال وجـود أكـثر مـن مهمـة جاهزة للتنفيذ، بينما الموارد الجاهزة لا يمكن استيعابها. وإذا وجدت مهمتان أو أكثر، مختلفة، في حال الجاهزية، فإنه سيختار المهمة ذات الأفضلية الأعلى، ليتمّ تنفيذهـا (وضعها في حالة Running). فإذا كانت المهام الجـاهزة ذات أفضليـات متسـاوية، أو أفضليات غير محـددة، فـإنّ الـترتيب الزمنـي غـير معـرّف فـي ADA، ويـترك القـرار للتنفيذ. وأيضاً، فإن خوارزمية الترتيب تكون خارج مجال اللغة، وتعريف المهام بلغـة

ADA، يتطلب بأن يكون الترتيب الزمني متوازناً (متوافقاً). لاحظ بأنّه يجب استخدام الأفضلية فقط، للإشارة إلى درجة الأهمية النسبية بين المهام. وينصح بعدم استخدام هذه الـ Pragma للحصول على تزامن المهام، لأن المعنى الدقيق للأفضليات المختلفة للمهام، يتعلق بقوة الآلة.

وليس من الضروري أن نعرف مدخلاً أو أكثر في مهمة. ومثال ذلك ما يلي:

task Producer;

وفي هذه الحالة، تمّ تعريف مهمة لا تمتلك أي مسار إتصال مرئي؛ والتعريف مشابه لشرط وحدة البرنامج الرئيسي. ويمكننا اعتبار هذا النوع من المهام كـ *Actor tasks*، إذ أنّ هذا النوع من المهام، لا يقدم أي خدمة لأي وحدة برمجية أخرى، ولكن تبقى دائماً في حالة نشطة. وبالطبع، فإن المهمة التي من النوع Actor task، تستطيع أن تستدعي المداخل المرئية لمهام أخرى.

وهذا التصنيف، يبين بأنّ الإتصال بين المهام في لغة ADA غير متزامن. ومن جهة أولى، توجد مهام من النوع Actor task لا تمتلك مداخل؛ ومن جهة ثانية، توجد مهام غير فعالة من النوع *Server Tasks*، إذ أنها تمتلك مداخل، لكنها لا تستحضر مداخل من مهام أخرى. وبالطبع، تسمح لغة ADA بخلق حالة وسط. ويمكننا رؤية مهمةٍ تحتوي مداخل تستخدم من قبل مهام أخرى، وأيضاً، يمكنها أن تستخدم مداخل مهام أ خر (نوع من المحول).

وعدم التناظر هذا، بديهي في رؤية المهام Actor/Server. والمهمة المستدعية، يجب أن تعرف إسم المهمة المدعوة. ومن جهة أخرى، فإن المهمة المدعوة، لا تعرف إسم المهمة المستدعية. وبالرجوع لقواعد لغة ADA، فإنه يجب التصريح عن الأغراض قبل استخدامها. ويمكننا التفكير بأنّه من المستحيل استدعاء مداخل كل مهمة، من قبل المهمة الأخرى، بشكل متبادل. وبالفعل، ليست هذه هي الحالة. ولنعتبر المثال التالي:

```
task First_Task is
  entry Service;
```

```
End First_Task;
task Second_Task is
  entry Service;
End Second_Task;
task body First_Task is
Begin
   ...
   Accept Service ...
   ...
End First_Task;
task body Second_Task is
Begin
   ...
   Accept Service ...
   ...
End Second_Task;
```

وبما أنّ توصيف المهمة وجسمها يمكن فصلهما نصياً، فيمكن التصريح عن توصيف المهمتين قبل كل شيء، وبالتالي، فإن مداخل كلا المهمتين تكون مرئية بالنسبة لجسمي المهمتين. وهكذا يمكن لجسم المهمة First_Task، استدعاء المدخل Second_Task.Service، والعكس صحيح.

ويمكننا تلخيص مناقشتنا حول المدخل، بما ندعوه عائلة من المداخل. وبشكل أساسي، تعرف العائلة مجموعة مداخل متكافئة، مفهرسة بقيم متقطعة، مثل فهرسة المصفوفة. وعلى سبيل المثال:

```
type Importance is (Low, Medium, High);
task Message is
  entry Get(Importance)(A_Message : out Message_Type);
  entry Put(Importance)(A_Message  : in   Message_Type);
End Message;
```

ويمكننا الرجوع إلى مداخل محددة، كما يلي:

```
Your_Message, My_Message : Message_Type;
```

```
Message.Get(High)(Your_Message);
Message.Put(Low)(My_Message);
```

وفي مقطع مقبل، سنناقش كيفية الإستخدام المشترك لعائلة من المداخل، كمهام تخدم الجدولة الزمنية.

وحتى الآن، أشرنا فقط، لكيفية التصريح عن أغراض المهام. ولنعتبر ما يلي:

```
task Terminal_Driver is
  entry Get(A_Character : out  Character);
  entry Put (A_Character : in    Character);
End Terminal_Driver;
```

وبالفعل، إنّ هذا التصريح مكافئ لما يلي:

```
task type Anonymous_Type is
  entry Get(A_Character : out Character);
  entry Put(A_Character  : in   Character);
End Anonymous_Type;
Terminal_Driver : Anonymous_Type;
```

وهنا، يشير الإسم Anonymous_Type، لنوع غير مرئي لأي مستخدم، مثل، نوع المعطيات. فإن نوع المهمة يعرّف نموذجاً، يمكن استخدامه لخلق عدة نسخ من أغراض المهام. ويعتبر نوع المهمة، كنوع معطيات خصوصي ومحدود، وبالتالي، لا يمكن إسناد أي قيمة له، ولا يمكن فحصه من أجل المساواة وعدمها، وأيضاً، يمكن استخدامه كمعاملات من النوع in لبرامج جزئية، أو مداخل مهام.

وفقط، العمليات الممكن تطبيقها على أغراض من نوع المهمة، هي المداخل المعرفة في توصيف نوع المهمة. وفي المثال التالي، نعرّف نوع معطيات، ونصرّح عن أغراض منه:

```
task type Resource is
  entry Release;
  entry Seize;
End Resource;
Buffer : Resource;
Segment : array (1..100) of Resource;
```

و بالتالي، يمكننا الرجوع إلى مداخل من أغراض محددة، كما يلي:

```
Buffer.Relaze;
Segment(7).Size;
```

والمثال الأخير، يوضح عائلةً من المهام، وهو يختلف عن مثال يستخدم عائلةً من المداخل. ومن أجل عائلة من المداخل، نعرّف فقط، مهمةً واحـدة، بمجموعـة مـن المداخل المختلفة (مسـارات إتصـالات). وفـي حـال الغـرض Segment، قمنـا بـالفعل بتعريف مجموعة من عدة مهـام (١٠٠ مهمـة مختلفـة)، ولكـل مهمـة مدخليـن، الأول Release والثاني Seize.

ويمكننا أيضاً، إستخدام أنواع المهام في تصريحات أُخرى، ومثال ذلك ما يلي:

```
task type Resource is
  entry Release;
  entry Seize;
End Resource;
type Locked_Data is
   record
    Element : Integer;
    Key       : Resource;
  End record;
My_Data : Locked_Data;
type Heap is access Resource;
```

ومن أجل التصريح عن أغراض من نوع مهمة مثل My_Data، يتم التنشيط بعـد القواعد التي ذكرناها: فالتصريح أولاً، ومن ثـمّ لا يتـم تنشـيط غـرض المهمـة، إلا فـي نهاية منطقة التصريح المغلقة. ويمكن تسمية مداخلها، كما يلي:

```
My_Data.Key.Release;
My_Data.Key.Seize;
```

وكما في مثال «الكومة» (Heap) المشار إليه، مـن المفيـد تعريـف نـوع وصـول، على نوع مهمة، وذلك في حال عدم معرفة العدد الفعلي للمهام التي نحن بحاجة لها، أو إذا أردنا تغيير تعاريف (Identities) المهام. وهنا فإن قواعد التنشيط مختلفة: حيث

يتم تنشيط المهمة، عند تنفيذ المخصص (Allocator). والترميز التالي، يبين كيفية عمل هذا النوع من التنفيذ:

```
Pool : Heap;
Pool := new Resource;  -- task is activated
```

وبالتالي، فإننا نستطيع استدعاء مدخل من المهمة، كما يلي:

```
Pool.Release;
Pool.Seize;
```

وكخاتمة لمناقشة توصيف المهام، لاحظ وجود عدة واصفات (Attributes) للأغراض، ولأنواع المهام. وهذه الواصفات، ما يلي:

```
T'Address      -- Start address for the task.
T'Callable -- True When T is not completed or terminated.
T'Size          -- Storage Space needed for T.
T'Storage_Size -- Storage space reserved for an activation of T.
T'Terminated    -- True when T is no longer active.
```

وفي فصل لاحق، سنناقش وبالتفصيل، كل من هذه الواصفات.

أجسام المهام (Task Bodies):

بكل توصيف مهمة، يرتبط جسم مهمة، يعرّف عمل المهمة. وإن شكل جسم مهمة، مشابه تماماً لجسم برنامج جزئي، فهو يتألف من قسم تصريحات، متبوع بسلسلة تعليمات، ومعالج إستثناءات إختياري. وعند تنشيط المهمة، يتم إنجاز العناصر (Items) المتواجدة في قسم التصريحات، وبعد ذلك، يتم البدء بالأعمال المرتبطة بسلسلة التعليمات. وبما أننا غالباً، نريد للمهمة أن تتابع المعالجة بشكل غير محدد، فإنه يمكن للتعليمات أن تأخذ شكل حلقة لا نهائية. فعلى سبيل المثال، يمكننا تعريف مهمة مراقبة (watchdog) بسيطة، كما يلي:

```
task Water_Monitor;
task body Water_Monitor is
Begin
 Loop
```

```
  if Water_Level > Maximum_Level then
   Sound_Alarm;
  End if;
  delay 1.0;
 End Loop;
End Water_Monitor;
```

وستستمر هذه المهمة بشكل دائم، وتصدر تنبيهاً عندما يزداد مستوى الماء فـوق حد معيـن (Maximum_Level). والتعليمـة Delay، تجعـل المهمـة تنتظـر (وهـو شـرط يحدده مطوّر البرنامج، ليصف حالة نوم) على الأقل ثانيـة واحـدة، وبعـد ذلـك تتكـرر الحلقة بسرعة، ليست أقل من ثانية واحـدة. وبـالفعل، يمكننـا السماح لهـذه المهمـة بالوصول للأغراض العامة Water_Level وMaximum_Level، وخاصـة إذا اسـتخدمت بقية المهام هذه الأغراض. وسنناقش مسألة المتغيرات المشتركة، فـي قسـم لاحـق مـن هذا الفصل.

فإذا عرفنا مداخل لمهمة، يجب على جسم المهمة أن يحتوي على الأقل تعليمة Accept واحدة، موافقـة لكـل مدخـل. وعلـى سـبيل المثـال، لنعتـبر توصيـف المهمـة التالي:

```
task Consumer is
  entry Transmit_Message(A_Message : in String);
end Consumer;
```

والترميز التالي، يطابق جسم المهمة:

```
task body Consumer is
Begin
  Loop
    accept Transmit_Message(A_Message : in String) do
      Text_IO.Put(Modem, A_Message);
     End Transmit_Message;
  End Loop;
End Consumer;
```

ومن أجل مهمةٍ من النوع nonactor، ومن أجل تحقيق موعد، يجب أن يتحقق الشرطان التاليان:

- إستدعاء مدخل من خارج المهمة.
- وتعليمة accept، الموافقة من داخل جسم المهمة.

وبالموافقة مع قواعد لغة ADA الخاصة بالموعد، إذا كانت المهمة Task_1 جاهزة للموعد قبل المهمة Task_2، فإنّ المهمة Task_1 ستنتظر حتى تصبح المهمة Task_2 جاهزة. وبشكل عام، ستضع مهمة نفسها في حالة نوم، بينما يتم تنفيذ مهمة من النوع *busy wait*. ونقول عن مهمة بأنها *busy wait*، إذا استخدمت منابع المعالجة، دون إجراء أي عمل مفيد، بينما تنتظر حدث. ونعني بـ *sleeping wait*، بأنّ المهمة مازالت تنتظر حدثاً، ولكن الإجراء أصبح معلقاً، حتى حدوث الحدث المنتظر. وعندما يبدأ موعد، يتم تنفيذ سلسلة التعليمات المرتبطة بتعليمة accept. وعندما تُكمِل المهمة المخدمة تنفيذ تلك التعليمات، يكتمل الموعد، ويتم تحرير المهمتين لتستمر بالعمل على التوازي.

ومن أجل كل مدخل، يستطيع جسم مهمة أن يحتوي تعليمة أو أكثر من accept. ويجب أن تظهر تعليمة accept مباشرةً في جسم مهمة؛ ولكن وفقاً لقواعد لغة ADA ، لا يمكن إنجاز ذلك داخل برنامج جزئي آخر، على سبيل المثال. وهذه القاعدة ضرورية لمنع مهمة من تنفيذ تعليمة accept، التابعة لمهمة أُخرى. وتعليمة accept، مؤلفة من كلمة محفوظة وهي accept ، متبوعة بإسم مدخل مع أدلة إختيارية (من أجل عائلة من المداخل)، ومن جزئها الصوري إذا وجد. والأعمال المرتبطة بالموعد تأتي فيما بعد، ومحددة بالعبارة

accept do.. end . وفي حال عدم وجود أي عمل، نستطيع إهمال عبارة ..do.. end.

ومثال ذلك ما يلي:

```
task Sequencer is
  entry Phase_1;
```

```
  entry Phase_2;
  entry Phase_3;
End Sequencer;
task body Sequencer is
  accept Phase_1;
  accept Phase_2;
  accept Phase_3 do
    Initiate_Launch;
  End Phase_3;
End Sequencer;
```

وهنا أجبرنا ترتيب إتصالاتنا بين المهام. ويجب إجراء موعد لـ Phase_1، ومن ثمّ لـ Phase_2، من ثمّ لـ Phase_3، حتـى نستطيع تنفيـذ التعليمـة Initiate_Lunch. وخلال المواعيد الوسيطة (Intermediate)، لا يمكن تنفيذ أي عمل، إلاّ تزامن المهام.

١٤ ـ ٢ ـ تعليمات المهام (Task Statements):

بمـا أنّـه بالأسـاس، تـمّ تصميم لغـة ADA مـن أجـل التطبيقـات ذات الزمــن الحقيقي، فمن المفهوم أن تحتوي بُنى للتعبير عـن أحـداث بالنسبة لـلزمن. وبشـكل خاص، فإن تعريف لغة ADA يحتـوي علـى حزمـة برمجيـة تُدعـى Calendar، التـي تصـدر النـوع Time، والتـابع المعـرّف مسـبقاً Clock، الـذي يعيــد الوقــت الســائد. وبالإضافة لذلك، فـإن الحزمـة البرمجيـة Standard، تحتـوي علـى النـوع Duration، المعرّف مسبقاً، والذي يستخدم للتعبير عن وحدات الثوانـــي. ويمكننـا اسـتخدام تعبيـر بسيط يزود قيمة من النوع Duration في تعليمة Delay، كما يلي:

```
delay 10.0;          -- Delay 10 Seconds
delay Next_Time - Calendar.Clock;-- Delay For Some Delta Time
```

وفي مثالنا هذا، فإن Calendar.Clock يمثل استـدعاء التـابع Clock، الموجـود في الحزمة البرمجية Calendar، والذي يعيـد الوقـت السـائد. ويتمثـل تـأثير التعليمـة Delay، بتأخير (تنويـم) كـل تنفيـذ فـي الوحـدة، علـى الأقـل، مـدة المجـال الزمنـي

المحدد. ويمكننا الإستمرار بتعريف الثوابت Seconds, Minutes, Hours، لجعل تعليمـة delay أكثر قابليةً للقراءة، كما يلي:

```
Seconds : constant Duration := 1.0;
Minutes  : constant Duration := 60.0;
Hours     : constant Duration := 3600.0;
delay 2*Hours + 7*Minutes +36*Seconds;
```

بينما تعتبر التعليمة التالية غير صحيحة، بسبب عدم تطابق الأنواع:

```
delay 4.5*Minutes;   -- Illegal!
```

فإنّ التعبير 4.5*Minutes، يمثل جـداء قيمـة حقيقيـة عامـة، مـع قيمـة ممثلـة بالفاصلة الثابتـة، ليكـون نـاتج الجـداء قيمـةً حقيقيـةً عامـةً ثابتـة، ليسـت مـن النـوع Duration. وإن استخدام تحويل الأنواع أو التصريح عن Minutes كثابت صحيح، يمثل حلاً لهذه المشكلة.

لاحظ بأننا لم نقل أنّ التعليمة Delay توقف المعالجة لمـدة معينـة مـن الزمـن، ولكن بالعكس، فهي تعرّف مقداراً من التأخير أصغرياً. فإذا كـان هنـالك حـلاً يحتـوي على عدة مهامٍ تُنفذ على معالج واحد، فإنّ المهمة الجاهزة للتنفيذ، يمكنهـا أن تنتظـر منابع المعالجة. والتأخير الفعلي، يمثل التأخير الأصغـري، مضافـاً إليـه الزمـن الـلازم لإيقاظ المهمة التي كانت نائمة. وعلى أية حال، ففي حال تطبيق محمـول، يمكـن أن نحتاج لإجراء يأخذ لحظات زمنية دقيقة، مثل أ خذ عينات من حساس.

وفيما يلي، مثال يستخدم تعليمـة Delay بشـكل غـير صحيـح، للحصـول على حلقة غير نظامية:

```
-- an invalid algorithm
Loop
-- Some Timed Action
  delay 30*Seconds;
End Loop;
```

ففي هذه الحالة، ستستغرق الحلقة على الأقل ٣٠ ثانية لتكتمل، بالإضافة لزمن تنفيذ التعليمات. وعلى أية حال، لا يمكننا التأكيد بأنّه سـيبدأ تنفيـذ الحلقـة كـل ٣٠

ثانية، والسبب في ذلك، عدم تحديد الترابط بين تزامن المهام، وإمكانيــة وجـود عـدة إجراءات بحاجة لنفس منابع المعالجة.

وكأفضل حل، هو حساب الزمن اللازم لبدء الحلقة الثانية، ثم بعد ذلـك إجـراء التأخير الزمني الضروري فقط للوصول إلى هــذا الزمـن. وبهـذه الإسـتراتيجية، سـنكون متأكدين بأنّ الحلقة، بشكل وسطي، ستأخذ قيماً زمنية دقيقة عند التنفيذ.

```
Declare
 Seconds : constant Duration :=1.0;
 Time_Interval : constant Duration :=30*Seconds;
 Next_Yime : Calendar.Time := Calendar.Clock;
Begin
 Loop
  delay Next_Time - Calendar.Clock;
  -- Some Actions Taking Less Than Time_Interval To Process
  Next_Time := Next_Time + Time_Interval;
 End Loop;
End;
```

وكما سنرى في فصل مقبل، هناك طريقة أفضل للحصـول علـى حلقـة نظاميـة، تتمثل باستخدام المقاطعة الزمنية.

وحتى الآن، قمنا بفحص العناصر الأساسية، من تنشيط المهام، وإنهائها، ودلالات مواعيـد المهـام. ولقـد حققنـا إتصـالاً بسـيطاً بيـن المهـام، مـع زوج مـن البُنـى الممثلـة بــ entry/accept . ونقول بأنّ الإتصال بسيط، لأنّ المهمة التي تنتظر نقطة الموعـد فـي البدء، يجب أن تنتظر وصول البقية بنفس النقطة. وعلى أية حال، فإن فراغ مسألتنا ليـس غالبـاً، بسيط تماماً. وعلى العكس، يمكننا الاهتمام بـالزمن الفعلـي للحـدث وبـدل السـماح للمهمـة بالإنتظار وقتاً غير محدد، نريد أن تترك المهمة محاولـة تحقيـق الموعـد، بعـد مـدة زمنيـة معينة. وأكثر من ذلك، فإن مهمةً واحدة، يمكنها الإختيار من بيـن عـدة مداخـل مختلفـة، مستدعاة من مخدمات طلبات المستخدمين. وتـزّود لغـة ADA ببعـض التعليمـات الخاصـة بمعالجة مثل هذه الحالات، من إتصال المهام.

ويمكن تلخيص مختلف صفوف الإتصال بين المهام بلغة ADA، فيما يلي:

- إتصال بسيط.
- موعد مختار من قبل المخدّم.
- موعد مختار من قبل المستدعي.

وسنقدم تعليمات لغة ADA لكل شكل من أشكال الإتصال، باستخدام محاكاة الزبائن وعملاء البنك. إنّ فلسفة إدارة المهام بلغة ADA، توافق نموذجاً من إتصال الإنسان. ولهذا، أخذنا هذا المثال غير الصوري أولاً.

ومن الواضح، أن زبائن وعملاء البنك، يمثلون مهام مستقلة متزامنة في بعض النقط الزمنية. وخلال اليوم، يكون الزبون ملتزماً ببعض النشاطات، مثل الإستيقاظ، وتناول طعام الإفطار، والذهاب للعمل، ومن الممكن، والذهاب للبنك لإجراء بعض الحسابات. والعميل أيضاً، يستيقظ في الصباح، ويتناول طعام الإفطار، ولكن في بعض الأحيان، ليس في نفس الوقت، أو في نفس المكان، مثل الزبون. ومؤخراً، سيذهب العميل للعمل، وينتظر الزبائن. وهذان الكيانان، الزبون والعميل، يتفاعلان عند وصول الزبون إلى البنك، لإجراء بعض المعاملات. وفي البدء، دعنا نفترض أنّ العميل دقيق جداً، وبالتالي، سينتظر مجيء الزبائن. ويمكننا تجريد هذه المسألة، كما يلي:

```
-- Teller task
accept Make_Deposit(Id : in Integer; Amount : in Float) do
  Balance(Id) := Balance(Id) + Amount;
 end Make_Deposit;
-- Customer task
Teller.Make_Deposit(Id => 1273, Amount => 1.0);
```

هذا مثال عن الإتصال البسيط بين المهام بلغة ADA . لاحظ كيف سمينا مدخل المهمة Make_Deposit، لتشير لعمل مجرد. فإذا حصل الزبون على استدعاء المدخل، فإنه بالتالي، سينتظر حتى يقبل الزبون الرسالة. وبالعكس (Conversely) ، إذا حصل العميل على تعليمة accept قبل جاهزية الزبون لموعد، فسينتظر العميل حتى يصل الزبون لإجراء الحساب. وفي حال وصول المهمتين لموعد، عندها سيتم تنفيذ سلسلة

التعليمات المرتبطة بتعليمة المخدم (العميل) accept، والمحصورة بين do .. end، ويتـم تبادل الرسائل.

ففي هذا المثال، يغير العميل ببساطة، الميزانية المطابقـة. وفـي حـال اسـتدعاء عدة زبائن لـ Make_Deposit، سيدخلون رتل الإنتظار المعرّف ضمنيـاً مـن أجـل ذلـك المدخل الخاص، وفق الترتيب، الـذي تـمّ وفقـه اسـتدعاء العميـل. وإذا وصـل الزبـون الرئيس (ذو الأفضلية العالية) بعد الزبون الوسطي (ذي الأفضليـة المنخفضـة)، فسـيتم تخديم الزبون الوسطي أولاً من قبل العميل، لأنه وصل قبل الزبون الرئيس.

وإن هذا التجريد، لا يصف العميل الأكثر فعالية: إذا كان هنالك عدد قليل مـن الزبائن خـلال اليـوم، فـإنّ العميـل يمضـي معظـم وقتـه فـي إنتظـار الزبـائن، لإجـراء Make_Deposit. ولا يسمح لنا الإتصال البسيط للمهـام، بالحفـاظ علـى نشـاط العميـل. لذلك، يجب إسناد عدة أشياء للعميل، للقيام بتنفيذها، مثل، تخديم زبـائن يراجعـون من أجل السيارات. ويمكن معالجة هذه المسألة كمدخـل مهمـة آخـر، وبالتـالي، فـإن ترميز العميل Teller، يصبح كما يلي:

```
task Teller is
  entry Make_Deposit(Id : in Integer; Amount : in Float);
  entry Make_Drive_Up_Deposit(Id:in Integer;Amount:in float);
end Teller;
task body Teller is
begin
 Loop
  Select
   accept Make_Deposit(Id : in Integer; Amount : in Float) do
     ....
   end Make_Deposit;
  or
   accept Make_Drive_Up_Deposit(Id:in Integer; Amount:  in Float) do
     ....
   end Make_Drive_Up_Deposit;
  End Select;
 End Loop;
end Teller;
```

وبلغة ADA فإننا ندعو هذا التركيب بـ (إنتظـار مختـار) Selective wait. ففـي هذه الحالة، يستطيع العميل إختيار مدخل من مدخلين ممكنين، أو ينتظر حتى يجهز أحد المدخلين للتخديم. وفي قمة تعليمة Select ، يفحص العميل أرتـال الإنتظـار مـن أجل كل مدخل (بواسطة آلية إدارة المهام). فإذا لم يكن هنالك زبائن ينتظرون موعداً، عندها سينتظر العميل على قمة تعليمة Select. وعندما يحـدد العميـل إمكانيـة تحقيـق موعد آني مع زبـون استدعى Make_Deposit أو Make_Drive_Up_Deposit، عندهـا يختار العميل ذلك المدخل. وفي حال استدعاء الزبون لكلا المدخلين في نفس الوقت، عندها يختار العميل أحد المدخلين للتخديم. ويجب ألا نربط تسلسل الإختيارات، لأنّ ترتيب المهام غير محدد. وإضافة لذلك، فإن الترتيب الذي وضعت به تعليمة accept، لا يؤثر على إختيار المهام.

وعلى أية حال، فعند التنفيذ، يتم الإختيار وبحرية، لأية تعليمـة accept. ومـن حقنا أن نختار إحدى التعليمات حسب ترتيبها. وإن أفضل طريقة صالحة للإستخدام، هي طريقة الترتيب الحلقي.

ويبقى هذا غير جيد بشـكل كـاف للعميـل. ففـي حـال عـدم حـدوث معـاملات مطلقاً، فإننا نرغب أن يبقى العميل مشغولاً، وربمـا عمـل ببعـض الملفـات. وبالتـالي، يمكننا إضافة عبارة else للحل السابق، للحصول على هذا النشاط:

```
task body Teller is
begin
 Loop
  Select
   accept Make_Deposit(Id : in Integer; Amount : in Float) do
     ....
    end Make_Deposit;
  or
accept Make_Drive_Up_Deposit(Id : in Integer; Amount : in Float) do
     ....
    end Make_Drive_Up_Deposit;
   else
```

```
    Do_Filing
  End Select;
   ....
 End Loop;
end Teller;
```

وتعرف هذه البنية بـ "إنتظار إختياري مع جزء وإلا" (Selective wait with an else part). ففي هذه الحالة، إذا لم يستطع العميل إجراء موعدٍ مباشرةً، عندها سـيقوم بتنفيذ سلسلة التعليمات المشار إليهـا بعـد else. وفـي غـير ذلـك، فـإن دلالـة تعليمـة select هذه، تكون مطابقةً لما سبق. ولاحظ عـدم إمكانيـة مقاطعـة التعليمـات الخاصـة بالجزء else. فحتى لو طلبت عدة مهـام للتخديـم، فسـوف ينتهـي أولاً تنفيـذ سلسـلة التعليمات التابعة للجزء else، ومن ثمّ، الإنتقال إلى بداية الوحدة.

ومازلنا نستطيع إنجاز أفضل من ذلـك. فعلـى سـبيل المثـال، لا يسـمح العميـل للزبون بإجراء تغيير في حسابه، إلا خلال ساعات عمل البنك (ساعات محدودة). ومن أجل هذا المثال، سنفترض بأنّ ساعات العمل النـافذة، الخاصـة بالسـيارات مختلفـة. وللإشارة بأنّ هذه الخدمات يمكن أن تكون مؤقتاً غير صالحة، يمكن التعبير عن ترمـيز العميل كمايلي، باستخدام ما يُدعى بالحارس:

```
task body Teller is
Begin
 Loop
  select
   when Banking_Hours =>
   accept Make_Deposit(Id : in Integer; Amount : in Float) do
       ....
     end Make_Deposit;
      or
       when Drive_Up_Hours =>
   accept Make_Drive_Up_Deposit(Id:in Integer; Amount : in Float) do
        ....
       end Make_Drive_Up_Deposit;
```

```
      else
        Do_Filing;
   end select;
    ....
 End Loop;
End Teller;
```

وتُدعى هذه البنية، بالإختيار مع حراس. فإذا لم يمتلك حارس أحد المداخل الإختيارية، أو إذا تمّ تقييم التعبير المنطقي على حارس خاص (المشار إليه بعبارة When) بـ True، يمكننا اعتبار accept الخاصة بهذا المدخل مفتوحة. وإذا تمّ تقييم التعبير المنطقي بـ False، فإنّ accept الخاصة بهذا المدخل مغلقة، وحتى أنه لا يمكننا اعتبارها من أجل الإختيار؛ وهي شبيهة بحالة عدم وجود المدخل. وبالطبع، يمكن لحالة الحارس أن تتغير مع الزمن، كما هي الحال عليه في مثالنا. ولاحظ عدم التقييم المستمر للحراس؛ إذ أنّ تقيم الحارس يتم فقط، عندما يكون على قمة تعليمة select، في طريقنا لحساب العمل التالي. ويمكن استخدام الحراس فقط، من أجل تعليمات accept، وليس من أجل الجزء الخاص بـ else. وبالإضافة لذلك، إذا كانت جميع المداخل مغلقة، ولا وجود للجزء else، في هذه الحالة، سيظهر الإستثناء Program_Error ليشير إلى خطأ، لأنّه من أجل كل عميل نشط، يجب أن يسند له عمل.

ولنخطو أكثر من ذلك. فمن البديهي، أن العميل يتمثل بإنسان فقط، ولا يستطيع إنتظار الزبائن إلا في وقت محدد. وفي حال ملاحظة العميل لعدم وصول زبائن خلال ساعتين من الزمن، فإننا نريد عندها، بأن يكون العميل قادراً على إنجاز استراحة. ويمكن أن نعبر عن ذلك، كما يلي:

```
task body Teller is
Begin
 Loop
 select
  when Banking_Hours =>
  accept Make_Deposit(Id : in Integer; Amount : in Float)  do
```

```
     ....
    end Make_Deposit;
  or
  when Drive_Up_Hours =>
     accept Make_Drive_Up_Deposit (Id : in Integer;
                                   Amount : in Float) do
       ....
     end Make_Drive_Up_Deposit;
   or
    delay 2*Hours;
    Take_A_Break;
 end select;
 ....
End Loop;
End Teller;
```

وندعـو هـذا الشـكل مـن المواعيـد المختـارة بــ "الإختيـار، مـع فـتـرة تـأخير إختيارية". وفي مثالنا، إذا لم ينجز العميل أي موعد مع أيّة مهمة خلال ساعتين على الأقل، فإنه سيتم تنفيذ سلسلة التعليمات التي تلي delay. وكما هو الحـال مـع الجـزء else، فلا يمكن إجراء مقاطعة على التعليمات التالية لـ delay. وتشير قواعد لغة ADA بأنّـه يمكـن لعـدة تخيـيرات مـن تعليمـة select، أن تملـك جـزء تـأخير delay. ومـن الواضح،أنه سيتم إختيار عدة مداخل، بأقصر زمن تأخير دائماً قبـل البقيـة. ولا يمكـن لتعليمة "الإختيار مع فترة تأخير إختيارية" أن تمتلـك جـزء else، لأنّـه لـن يتـمّ أبـداً إختيار التأخير الإختياري.

وبغض النظر عن هذه الخصوصية، يمكننا تشكيل جميع الأشكال المختلفـة مـن الإتصال حسب حاجتنا، متضمناً ذلك:

- الإختيار مع وإلا (select with an else) .
- الإختيار مع وتأخير (select with delay) .
- الإختيار مع حراس (select with guards) .
- الإختيار البسيط (simple select) .

فإذا أردنا أن يُنجز العميل فقط Make_Deposit، ويأخذ استراحة بعد مضي ٣٠ دقيقة من الإنتظار، نستطيع إنجاز الحل بلغة ADA، كما يلي:

```
task body Teller is
Begin
 Loop
  select
  accept Make_Deposit(Id : in Integer; Amount : in Float) do
     ....
    end Make_Deposit;
   or
     delay 30*Minutes;
     Take_A_Break;
  end select;
   ....
 End Loop;
End Teller;
```

ويجب أن نملك بعض الطرق لإنهـاء العميـل. وتوجـد ثـلاث طـرق لحـل إنهـاء المهام. أولاً، تشير قواعد لغة ADA، بأنّه لا يمكن أن تنتهي مهمة بشكل طبيعي، إلا إذا وصلت إلى نهاية سلسلة تعليماتها، وإنتهاء جميع المهام المرتبطة بهـا، أو تكـون جاهزة للإنتهاء. وبمعنى آخر، يجب على جميع المهام، إعتباراً من تصريحٍ محدد، أن تكون منتهية، أو جاهزة للإنتهاء معاً. ومثال ذلك، ما يلي:

```
task Outer;
task body Outer is
 task Inner;
 task body Inner is
 Begin  -- Inner
   -- sequence of statements
 End Inner;
Begin    -- Outer
   -- sequence of statements
End Outer;
```

فتكتمل المهمة Outer، عندما تُنهي تنفيذ سلسلة التعليمات الخاصة بها. وعلى أية حال، لا تنتهي المهمة Outer، حتى تنتهي المهمة الإبـن Inner أيضاً، أو تكون جاهزة للإنتهاء. وتملك هذه القواعد بعض المقترحات المهمة، عندما تتعامل مع برامج جزئية. فلنفترض أننا صرّحنا عن المهمة A_Task، داخل البرنامج الجزئي A_Subprogram، بدلاً من وحدة مكتبية:

```
procedure A_Subprogram is
 task A_Task;
 task body A_Task is
 Begin
   -- Sequence of statements
 End A_Task;
Begin  -- A_Subprogram
 -- Sequence of statements
End;   -- A_Subprogram
```

وبما أنّ البرنامج الجزئي A_Subprogram قد أنهى سلسلة تعليماته، فإنّ قواعد لغة ADA تتطلب ألاّ نعود من البرنامج الجزئي A_Subprogram، حتى تنتهي المهمة A_Task.

وفي حال المهام التي تطرح عدة مداخل، فمن التقليدي أن نرى أجسام المهام على شكل حلقة تتضمن تعليمة select، بالمقارنة مع جسم المهمة Teller. وهنا، فإن درجة الإتصال بين المهام معقدة كثيراً، فمن العمومية كتابة جسم مهمة بطريقة تنهي تنفيذ سلسلة تعليماته بشكل طبيعي. ومن أجل هذا السبب، ومن أجل السماح بإنتهاء مناسب بحضور مهام أبناء معقدة، تسمح لغة ADA باستخدام الإنتهاء (terminate) الإختياري في داخل تعليمة select. وفقط، تعليمة إنتهاء واحدة ضمن تعليمة select، مسموحة. وعلى أية حال، لا يمكن ربطها مع تعليمة delay، أو else الإختيارية. وفيمايلي، مثال عن الإنتهاء الإختياري:

```
task body Teller is
Begin
 Loop
```

```
 select
  accept Make_Deposit(Id : in Integer; Amount: in Float) do .
    ....
   end Make_Deposit;
  or
    terminate;
 end select;
  End Loop;
End Teller;
```

ففي هذه الحالة، يمكن ضمنياً إختيار تعليمة terminate الإختيارية، الموجودة في المهمة Teller. ويمكن أن تنتهي المهمة فقط، عندما تكون المهمة الأب جاهزة للإنتهاء، وعندما تكون جميع المهام المرتبطة بـ Teller أيضاً، جاهزة للإنتهاء. وتكون المهمة الأب جاهزة للإنتهاء، عندما تكون منتظرة عند terminate، أو منتظرة عند العبارة End النهائية، الخاصة بها. وبدون شك، تعتبر هذه الطريقة أكثر الطرق الملائمة لإنهاء المهام، خصوصاً، إذا كانت هناك عدة مهام في الحل. وكقاعدة عامة، فإن تطبيق تعليمة terminate لأي مهمة مخدمة، يصل إلى هذا الإنتهاء الملائم. فإذا أنهت جميع المهام في مجموعة ما أعمالها، وأصبحت جاهزة لقبول تعليمات terminate الخاصة بكل مهمة، فإنّ جميع هذه المهام تنتهي مع بعض.

وتسمح قواعد لغة ADA من خلال تعليمة select، تضمين تعليمة terminate، أو تعليمات delay، أو عبارة else، أو لا شيء من التعليمات الثلاث. والتعليمات الثلاث السابقة الذكر (terminate, delay, else)، تمثل المنع المتبادل. فلا يمكننا رؤية تعليمة terminate، بنفس الوقت الذي نشاهد فيه وجود الجزء else، داخل نفس تعليمة select .

والطريقة الثانية لتنظيم إنهاء المهام، تتمثل بتزويد مدخل يتم استدعاؤه عندما نريد صراحة إنهاء مهمة. فعلى سبيل المثال، يمكننا إضافة المدخل Shut_Down للمهمة Teller، وعند قبول الرسالة فقط، يتم الخروج من الحلقة. فإذا أصبحت الحالة

خطيرة، ولدينا عميل عاجز بالحقيقة، فيمكننا إنهاء مهمة محددة بشكل غير نظـامي، بتنفيذ التعليمة abort، كما يلي:

```
abort Teller;.
```

ويمكن توقيف مهمة، مادامت مرئية من نقطـة محـددة. وهـذه التعليمـة، تقتـل مهمة محددة، وجميع المهام المرتبطـة بهـا. ويعتـبر توقيـف مهمـة، مـن الطـرق غـير الملائمة لإنهاء مهمة، ولا ينصح باستخدامه، إلا في حال فشل بقية الطرق.

ولقتل مهمة، من المفضل محاولة إجراء موعـد بواسـطة المدخـل Shut_Down، قد تمّ تعريفه سابقاً من أجل تلك المهمة المخدمة، وإجراء تأخير زمني لفترة محـددة، ومن ثمّ توقيف المهمة. وعلى سبيل المثال ما يلي:

```
Teller.Shut_Down;
delay 30*Seconds;
abort Teller;
```

وهـو إعـلان للمهمـة Teller بأننـا سـننهيها (مـن خـلال اسـتدعاء المدخـل Shut_Dwon)، ومن ثمّ الإنتظار قليلاً. وتعرف هذه التقنية بـ giving the task its last wishes.

وحتى الآن، قد فحصنا المواعيد المختارة من أجل المخدم، ولكـن توجـد نقـاط مشابهة من أجل مهمة الزبون. فعلى سبيل المثال، سـينتظر الزبـون فـترةً مـن الزمـن، ليتم تخديمه من قبل العميل Teller؛ وبالتالي، نريد من الـ actor task، تـرك محاولـة الموعد بعد فترة زمنية عظمى. فعلى سبيل المثال:

```
-- Customer task
select
  Teller.Make_Deposit(Id =>1273, Amount => 1_000.0);
 or
  delay 10*Minutes;
  Take_A_Hike;
end select;
```

وتعرف هذه البنية باستدعاء مدخل محدود زمنياً (a timed entry call). ففي هذه الحالة، عندما نبدأ تنفيذ تعليمة select هذه، ننتظر موعداً مع العميل Teller. فـإذا لـم يُخدم الزبون خلال ١٠ دقائق، نلغي طلبنا للمدخل Make_Deposit مـن رتـل الإنتظـار (هذا يغير الواصف Count في العميـل) . وعندهـا، ننفـذ سلسـلة التعليمـات المرتبطـة بجزء التأخير الزمني delay، ومن ثم، سيأخذ الزبون Take_A_Hike. وإذا تمّ تخديـم الزبون خلال ١٠ دقائق، بعد الموعد، يُتابع التنفيذ بعد نهاية تعليمة select.

وفي حال وجود زبون غير صبور، أي أنّه لا يُريد الإنتظار، فيمكـن التعبـير عـن هذه الحالة كما يلي:

```
-- Customer task
select
  Teller.Make_Deposit(Id =>1273, Amount => 1_000.0);
 else
   Run_Away;
end select;
```

وندعو هذه البنية باستدعاء مدخل شَرطي (a conditional entry call). إن هـذه البنية، دلالياً، مكافئة لاستدعاء مدخل محدود زمنياً، بتأخير زمني معدوم. ولكـن لغـة ADA تتضمن شكلاً مخالفاً للإشـارة صراحـة عـن غيـاب التـأخير الزمنـي. لاحـظ بـأنّ الإختلاف اللغوي حساس جداً بين هاتين البنيتين (else بدلاً من or).

وبهذا نكمل سلسلة التعليمات الخاصة للمهام. وحتى هذه النقطة، لـم نقـدم إلا حقيبة إجمالية من الأدوات؛ وفي المقطع التالي، سنفحص تطبيق المهام.

١٤ ـ ٣ ـ تطبيقات مهام لغة ADA
(Applications for ADA Tasks):

في فصول سابقة، وخلال فحوصاتنا للـبرامج الجزئيـة، والحـزم البرمجيـة، لـم نناقش فقط شكل كل واحدة من هـذه البنـى، بـل ناقشـنا تطبيقـات كـل منهـا. وإدارة المهام، تمثل وظيفةً لم يتم التطـرق لهـا فـي بعـض لغـات البرمجـة عاليـة المسـتوى، لذلك، من المحبذ تقديم استخدام مهام لغة ADA، علـى شـكل عـدة نمـاذج تطبيقيـة.

وبشكل عام، تمثل البرامج الجزئية نشاطات مجردة، كما تسمح لنا الحـزم البرمجيـة، بتجميع كيانات مترابطة منطقية. وعند الاستخدام الجيد للمهـام، فـإن كـلا البنيتيـن، ستعكسان بنية فضاء المسألة. ونفس الشيء صحيح بالنسـبة للمهـام. حيـث يجـب أن تعكس المهام، طبيعة مختلف الإجراءات في مجال المسألة.

ويمكن تقسيم تطبيقات مهام لغة ADA إلى أربعة مناطق، وهذه المناطق مايلي:

- الأعمال المتوازية (Concurrent Actions) .
- توجيه الرسائل (Routing Messages) .
- إدارة الموارد المشتركة (Managing Shared Resources) .
- معالجة المقاطعات (Interrupt handling) .

وسنعرض كلاً من هذه التطبيقات، مع أمثلة، في المقاطع التالية:

الأعمال المتوازية (Concurrent Actions) :

إن أحد الإستخدامات الأساسية لمهام لغة **ADA**، تتمثل بالتعبير عن الإجراءات المتوازية. مثلما نوهنا في بداية هـذا الفصـل، إن القليـل مـن لغـات البرمجـة، تسـمح بالتعبير عن التوازي. وإنّ لغة **ADA**، تحقق ما تمّ ذكـره بشـكل واضـح. ومثلمـا تابعنـا طرق التصميم غرضية التوجه، يمكن أن نجـد أفعـالاً، يمكـن أن تحـدث منطقيـاً علـى التوازي مع أفعال أخرى. وفي أعلى مستوى من حلّنا، يمكن تعليب الأعمال في داخل مهمة.

وعلى سبيل المثال، لنعتبر نظام سيارات حاسوبي، يراقب معـاملات المحـرك، مثل، ضغط الزيت، ودرجة حرارة الماء. فمن الواضح أن هذيـن العمليـن مسـتقلان عـن بعضهما. ويمكن أن تتطلب المسألة منبهاً صوتياً، عندما تصبح إحدى القيـم أعلـى مـن قيمة محددة. ويمكن عرض المهمتين في قسم التصريحات من البرنامج الرئيسـي، كمـا يلي:

```
task Oil_Monitor;
task Water_Monitor;
task body Oil_Monitor is
```

```
 Pressure : Float;
begin
 Loop
  Get(Pressure);
  if Pressure > Maximum_Pressure then
   Activate_Alarm;
  End if;
 End Loop;
End Oil_Monitor;
task body Water_Monitor is
 Temperature : Float;
begin
 Loop
  Get(Temperature );
  if Temperature > Maximum_Temperature then
   Activate_Alarm;
  End if;
 End Loop;
End Water_Monitor;
```

وبما أنّه لم تتم الإشارة في هذا المثال إلى الإسماء Max_Pressure وMax_Temperature ، فمن المحتمل أن يُعرّفا كقيمٍ ثابتة في البرنامج الرئيسي. ويمكن أن يكون Activate_Alarm، مدخل مهمة أخرى، أو يمثل ببساطة، البرنامج الجزئي الذي يشغل تحذيراً مرئياً أو مسموعاً. وفي جميع الحالات، لا توجد مهمة مراقبة لم تعرف مداخل، فلذلك، فإن كلاهما هي مهام فعالة (Actor task)، مثلما تمّ تعريفهما سابقاً. وإذا لم تحتو الآلة التي نعمل عليها إلا على معالج واحد، فتستطيع هاتان المهمتان أن تتقاسما استخدام المعالج (من الممكن وفق إستراتيجية تقاسم الزمن). ومن جهة أخرى، إذا وجدت عدة معالجات مثل المهام، فإنّ جميع المهام تعمل على التوازي. وعلى أية حال، تذكّر بأنّ كل تنفيذٍ، حرٌ بإختيار طريقته الخاصة بإدارة المهام التي لها درجة أفضلية متساوية. فمن الملائم أن يعطي التنفيذ منابع الحساب للمهمة Oil_Monitor، ولا ينفذ مطلقاً المهمة Water_Monitor ، بينما لم

تتوقف المهمة **Oil_Monitor** مطلقاً. ومن الواضح، أن لا يعتبر هذا التنفيذ مفيـداً. وإن أفضل تنفيذٍ، (وأيضاً أكثر ملاءمةً) يتمثل باستخدام التقسيم الزمني، وبالتالي، فإنّ كلاً من **Oil_Monitor** و**Water_Monitor**، تعمل لفترة زمنية محددة. وهكذا، يتـم تنفيـذ المهمة **Oil_Monitor**، خلال بضعة أجزاء مـن الميلـي ثانيـة – حتـى إذا لـم تتوقـف **(Block)** – ومن ثـم تشـير إلـى جاهزيتهـا. وعندئـذ، تعطـى منـابع المعالجـة للمهمـة **Water_Monitor**. ومن أجل الناقلية، من الضروري إعطاء قيم قصيرة للتأخير الزمنـي **Delay** في مثل هذه المهام، من أجل إجبار مهمة توقفت فجأةً، للتوقف لفـترة زمنيـة قصيرة، وهذا ما يسمح لمرتّب المهام، بجعل المهام الجاهزة بحالةٍ نشطة.

وليس من الضروري أن يقتصر اسـتخدام المهـام علـى الحلـول عاليـة المسـتوى. وبالعكس، يمكننا إيجاد خوارزميات بسيطة، تحتـوي مكونـات يمكـن أن تعمـل علـى التوازي. فعلى سبيل المثال، في جبر المصفوفـات، غالبـاً مـا يسـتخدم العلميـون فـي حلولهم للمسائل الهندسية، عدة مراحل من الحسابات تكون مستقلة، وبالتالي، يمكـن منطقياً معالجتها على التوازي. وإن إحـدى الحسـابات المشـتركة فـي الإيروديـنـاميك، تتمثل بضرب مصفوفةٍ مع شعاع. فلنفرض أننا نريد حساب الجداء التالي:

$$\begin{vmatrix} X(1,1) & X(1,2) & \dots & X(1,C) \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots \\ X(R,1) & X(R,2) & \dots & X(R,C) \end{vmatrix} * \begin{vmatrix} U(1) \\ . \\ . \\ . \\ U(C) \end{vmatrix}$$

ونتيجة هذه العملية، تتمثل بما يلي:

$$\begin{vmatrix} X(1,1)*U(1)+X(1,2)*U(2)+\dots+X(1,C)*U(C) \\ \dots\dots \\ \dots\dots \\ \dots\dots \\ X(R,1)*U(1)+X(R,2)*U(2)+\dots+X(R,C)*U(C) \end{vmatrix}$$

وإن ناتج ضرب مصفوفة عدد أسطرها R وعدد أعمدتها C، بشعاع عدد عناصره C، يتمثل بشعاع عدد عناصره R. وبدراسة الخوارزمية المستخدمة لتشكيل ناتج الضرب، نلاحظ بأنّه يمكن حساب كل سطر بشكل مستقل عن بقية الأسطر. وإن المهمة التي تحسب قيمة سطر وحيد، بسيطة تماماً.

والجزء الأول من الإستراتيجية الخاصة بالمهمة Partial_Product، ستكون بـ Send_Values التي تحتاج إليها في الحساب. ومن ثمّ، سنُنتج الجداء، وسنُعيد Recive_Value إلى المهمة الأب. ويمكن التعبير عن ذلك، كمايلي:

```
type Matrix_Row is array (Integer range <>) of Float;
type Pointer is access Matrix_Row;
task type Partial_Product is
  entry Recive_Value (Result : out Float);
  entry Send_Values  (First, Second : in Matrix_Row);
end Partial_Product;
Size : constant :=10;
type Matrix is array(1..Size) of Matrix_Row(1..Size);
A_Matrix      : Matrix;
Source_Vector : Matrix_Row(1..Size);
Product_Vector : Matrix_Row(1..Size);
task body Partial_Product is
  Product : Float;
  Vector_1 : Pointer;
  Vector_2 : Pointer;
begin
   accept Send_Values(First, Second : in Matrix_Row) do
     Vector_1 := new Matrix_Row'(First);
     Vector_2 := new Matrix_Row'(Second);
   end Send_Values;
   Product := 0.0;
   for Index in Vector1.all'Range
    Loop
     Product := Product + Vector_1(Index)*Vector_2(Index);
```

```
   End Loop;
  accept Recive_Value(Result : out Float) do
   Result := Product;
  end Recive_Value;
End Partial_Product;
```

ومـن أجـل هـذه المهمـة، قمنـا بتعريـف مدخليـن. المدخـل الأول، يتمثـل بــ Send_Values ، وهو يستخدم لنقل الشعاعين للمهمـة لمعالجتهمـا. بينمـا المدخـل الثاني، يتمثل بـ Recive_Value، وهو يستخدم للحصول على الجداء الجزئي. ولاحـظ كيفية إمكانية تطبيق هذه المهمة على مصفوفات من أيّ حجم: لقـد عرفنـا المعـاملات الصورية للمداخل لتكون أنواع مصفوفات غير مقيدة (من النوع Matrix_Row). ويتـم تحديدها عندما ترتبط مع المعـاملات الفعليـة، ويمكننـا العـودة لحدودهمـا مـن خـلال الواصف Range. وفي جسم المهمة، يجب علينا أولاً قبول المدخـل Send_Values ، ومن ثمّ نحفظ القيم First, Second بالغرضين Vector_1, Vector_2 من النوع المؤشـر. ولقد استخدمنا النوع المؤشر، لأننا لسنا متأكدين من حجم المصفوفة حتى زمن التنفيذ. ومن ثمّ، نحقق حلقة بطول كل شعاع، وبعـد ذلـك نحسـب الجـداء الجزئـي. ثـم بعـد تحقيق هـذا الجـداء، نقبـل المدخـل Recive_Value. ولنفـترض أنّ نـوع المهمـة هـذه مرئي، وبالتالي، فإن الشكل العام لخوارزمية ضرب مصفوفة بشعاع، يمكـن أن يكتـب كما يلي. سنفترض بأنّ A_Matrix، يمثل مصفوفة مربعة مـن النـوع Matrix، بحيـث أن كل سطر من النوع Matrix_Row وهو محدد بطول معين. بالإضافة لذلك، سنفترض بأنّ الشعاع Source_Vector، مرئي ومصـرح عنـه مـن النـوع Matrix_Row، وبنفـس طـول أسـطر المصفوفـة Matrix. وسـنصرح عـن الشـعاع Product_Vector ليحتـوي النتيجة. ويمكن تسمية هذه الأشعة صورياً، كما يلي:

```
declare
 Parallel_Product : array(Source_Vector'range) of Partial_Product;
Begin
 for Index in Source_Vector'range
  Loop
Parallel_Product(Index).Send_Values(A_Matrix(Index),Source_Vector;
```

```
  End Loop;
 for Index in Source_Vector'range
  Loop
Parallel_Product(Index).Recive_Value(Product_Vector(Index));
  End Loop;
End;
```

ففي هذا المثال، إن عائلة أغراض المهام (Parallel_Product) لا توجد إلا في مجال رؤية هذه الكتلة المحلية. وفي البدء قمنا بـ Send_Values لكل مهمة، بإرسال مُكوّن من A_Matrix، والمدخل Source_Vector. وبعد ذلك نحقق الحلقة حتى Recive_Value من كل مهمة. وبعد ذلك نضع ناتج الضرب الجزئي في المكون المطابق من Product_vector. لاحظ بأنّ هذه الطريقة خاصة، فقط، في حال احتواء النظام أكثر من معالج. وفي حال وجود معالج وحيد، سيوجد ضياع كبير في الزمن، بسبب الإنتقال من مهمة لأخرى. ويمكن تطبيق نفس التقنية على مجموعة معادلات مستقلة.

توجيه تدوير الرسائل (Routing Messages) :

إن مجال التطبيق النموذجي الآخر للمهام، يتمثل بتوجيه الرسائل؛ إذ يمكن أن نرغب بتوجيه رسائل لمهام أخرى، أو أجهزة فيزيائية أخرى. وبأبسط شكل لتوجية الرسائل، يمكن تعريف مهمة من أجل كل طرفي؛ وعندها تستخدم المهام لحجز المعطيات المنقولة. وعلى سبيل المثال، يمكن تخزين الخرج لطابعة. وعندما تضع مهمة أخرى محارف على الطابعة، فإنّ مهمة التخزين تقبل المدخل وتأخذ المحارف، لذلك، تستطيع المهمة الأولى تأمين استمرارية المعالجة. وعندها سترسل مهمة التخزين المحارف إلى الطابعة، في حال عدم وجود طلبات أخرى. فإذا لم تمتلئ حافظة التخزين، فإنّ المهمة تتقبل المحارف من أجل الطباعة. وعلى أية حال، إذا امتلأت الحافظة، تقوم عندها المهمة بإغلاق المدخل المختار، وتجبر مهام وضع المحارف لتنتظر، حتى تفتح الحافظة مرة ثانية.

وفي حل مسألتنا، اخترنا الحزمة البرمجية المولّدة Queues، التـي تـمّ خلقهـا في فصل سابق، من أجل تنفيذ حافظة داخلية باستخدام المحارف كعناصر. وبالإضافة لذلك، سنفترض بأنّ الخرج للطابعة صالح مـن خـلال ملـف يدعـى "Printer". ويمكـن كتابة الترميز، كمايلي:

```
task Spool is
  entry put(A_Character : in Character);
end Spool;
task body Spool is
   package Character_Queue is new Queues(Item => Character);
   use Character_Queue;
   Buffer : Character_Queue.Queue(Size =>100);
   A_Character : Character;
   Printer : Text_IO.File_Type;
Begin
  Clear(Buffer);
  Text_IO.Open(File     => Printer,
                         Mode => Text_IO.Out_File,
                         Name => "Printer");
   Loop
     select
       when Length_Of(Buffer) < 100 =>
         accept Put(A_Character : in Character) do
           Add(A_Character, Buffer);
         end Put;
      else
        if Length_Of(Buffer) > 0 then
          Remove(A_Character, Buffer);
          Text_IO.Put(Printer, A_Charcater);
       end if;
     end select;
   End Loop;
End Spool;
```

وبما أنّ قواعد لغة ADA لا تسمح بالتصريح عن المهام كوحدات مكتبية، فغالباً ما نُغلّف مهمةً داخل حزمة برمجية. وبعد ذلك نجعلها مرئيةً كالحزمة البرمجية، باستخدام كلمة with في سياق التوصيف. فعلى سبيل المثال، يمكننا كتابة توصيف حزمة برمجية من أجل المهمة Spool، كما يلي:

```
package Spooled_Print is
 Procedure Put(A_Character : in Character);
End Spooled_Print;
```

وبالتالي، يمكن كتابة جسم الحزمة البرمجية الموافق على الشكل التالي:

```
with Text_IO,Queues;
use Text_IO;
package body Spooled_Print is
task Spool is
 entry put(A_Character : in Character);
end Spool;
task body Spool is
  package Character_Queue is new Queues(Item => Character);
  use Character_Queue;
  Buffer : Character_Queue.Queue(Size =>100);
  The_Character : Character;
  Printer : Text_IO.File_Type;
Begin
 Clear(Buffer);
 Text_IO.Open(File     => Printer,
              Mode => Text_IO.Out_File,
              Name => "Printer");
  Loop
   select
    when Length_Of(Buffer) < 100 =>
     accept Put(A_Character : in Character) do
      Add(A_Character, Buffer);
     end Put;
   else
```

```
      if not Is_Empty(Buffer) then
        Remove(The_Character, Buffer);
        Text_IO.Put(Printer, The_Charcater);
      end if;
    end select;
  End Loop;
End Spool;
procedure Put(A_Character : in Character) is
Begin
  Spool.Put(A_Character);
End Put;
End Spooled_Print;
```

لاحظ بأننا استدعينا المدخل Spool.Put، من خلال الإجرائية Put. ووفق هذه الطريقة، فإن تنفيذ (زرع) (مهمة، أو برنامج جزئي) يكون مخفياً من المستخدم.

وفي مثال أكثر تعقيداً عن تمرير الرسائل، يمكننا الحصول على مهمة توجيه رسائل إلى مهام أخرى، كما هو الحال في نظام معالجة موزع. ويمكن لكل مهمة أن تكون actor/server لأنها تُعرف مداخل، ولأنها أيضاً تستدعي مداخل مهمة أخرى. وفي هذا المثال، سنفترض بأنّ المعلومات الممررة بواسطة هذه المهمة، غالباً ما تكون من النوع Message. وأكثر من ذلك، فإن مهمةً مستدعاة تستطيع توجيه رسالة لواحدة من ثلاث وجهات. وبالإضافة لذلك، تستطيع المهمة تعيين أفضلية الرسائل، بحيث أن الرسالة ذات الأفضلية العليا، يتم تخديمها أولاً. وبالتالي، فنحن بحاجةٍ لأن تكون الكيانات التالية، مرئيةً للمهمة:

```
procedure Main is
  type Message is ...
  My_Message, Your_Message : Message;
  type Priority is (Low, Medium, High);
  type Place is range 1..3;
  task Destination_1 is
    entry Send(A_Message : in Message);
  end Destination_1;
```

```
 task Destination_2 is
   entry Send(A_Message : in Message);
 end Destination_2;
 task Destination_3 is
   entry Send(A_Message : in Message);
 end Destination_3;
```

وقد افترضنا هنا بأنّ كل وجهة تعالج الرسالة بطريقة مختلفة. وإذا، كان بالعكس، تكون المعالجات متطابقة، ويمكننا استخدام عائلة من المهام.

ومن أجل مهمة التوجيه، فإنه تتطابق المعالجة من أجل كل مستوى أفضلية، وبالتالي، نستخدم عائلةً من المداخل. ويمكن أن يعرّف قسم توصيف المهمة، كما يلي:

```
task Transmit is
  entry Routed_Priority(Priority)(The_Message : in Message; To : in Place);
end Transmit;
procedure Route(The_Message : in Message; To : in Place) is separate;
task body Destination_1 is separate;
task body Destination_2 is separate;
task body Destination_3 is separate;
task body Transmit is separate;
```

لاحظ بأنّ استدعاء هذه المهمة يكون مقروءاً بشكلٍ جيد، بسبب اصطلاحات التسمية المستخدمة:

```
Begin
 Transmit.Routed_Priority(Medium)(My_Message,To => 1);
 Transmit.Routed_Priority(Low)(Your_Message,To => 3);
End Main;
```

وسيكون الأسلوب أفضل، بالتصريح عن نوعٍ مرقم من أجل Place، بدلاً من استخدام قيمة طبيعية.

ومن أجل جسم المهمة، ستستخدم تعليمة select في خوارزميتنا لتختار بين المداخل الممكنة. وبالإضافة لذلك، طالما توجد رسائل ذات أفضليات أعلى، ستغلق الإختيارات ذات الأفضليات الأدنى. ويمكن الحصول على ذلك باستخدام select مع

حراس. وبما أنّ خوارزمية إختيار الوجهة هي نفسها من أجــل كـل أفضليــة، ويمكننـا أولاً، كتابة البرنامج الجزئي التالي:

```
separate (Main)
procedure Route(The_Message : in Message; To : in Place) is
Begin
 Case To is
  when 1 => Destination_1.Send(The_Message);
  when 2 => Destination_2.Send(The_Message);
  when 3 => Destination_3.Send(The_Message);
 End Case;
End Route;
```

ولا توجد مشكلة باستدعاء عدة مهـام لهـذه الإجرائيـة، لأنّ لغـة ADA تتطلـب إعادة إدخال جميع البرامج الجزئية.

فإذا كان Route مرئياً لـ Transmit، فيمكننا بالتالي، إنجـاز جسـم Transmit، كما يلي:

```
separate (Main)
task body Transmit is
Begin
 Loop
  select
   accept Routed_Priority(High)(The_Message : in Message;
                        To : in Place) do
    Route(The_Message, To);
   end Routed_Priority;
   or
    when Routed_Priority(High'Count = 0 =>
    accept Routed_Priority(Medium)(The_Message:in Message;
                          To : in Place) do
      Route(The_Message, To);
     end Routed_Priority;
   or
```

```
    when Routed_Priority(High'Count = 0 and
          Routed_Priority(Medium)'Count = 0 =>
     accept Routed_Priority(Low)(The_Message : in Message;
                          To : in Place) do
      Route(The_Message, To);
     end Routed_Priority;
   end select;
  End Loop;
End Transmit;
```

لاحظ كيف استخدمنا الواصف Count، للتأكد من عدم وجود مهام ذات أفضليات عليا في حالة إنتظار. ودلالة Count تكون مساوية للصفر، في حال عدم وجود مهام تستدعي المدخل المعين.

إدارة الموارد المشتركة (Controlling Resources) :

عندما نقسم مسألة إلى أجزائها الوظيفية، بعد ذلك ننجز كل عمل باستخدام برامج جزئية. وبشكل عام لا ينصح بترك كيانات كهذه باستخدام معطيات عامة. ومثلما ناقشنا في فصل سابق، فإن استخدام المعطيات العامة، يزيد الإرتباط بين وحدات البرنامج، وهذا ما يجعل النظام أقل وثوقيةً وقابليةً للصيانة. ومن المفضل، فقط، تمرير المعطيات التي تستخدمها أو تغيرها وحدة برمجية ما. وفي حال المهام، على أية حال، يمكننا الحصول على حالة يوجد فيها مهمتان أو أكثر، بحاجةٍ للوصول إلى نفس المعطيات على التوازي. وبما أنّه يمكن لعدة مهام محاولة قراءة القيمة، بينما تحاول مهام أخرى كتابتها في نفس اللحظة، فإنه يجب أن نملك بعض الطرق لمنع جميع المهام، باستثناء واحدة، من الوصول إلى القيمة.

وإذا أرادت مهمتان أو أكثر، الوصول إلى كيان عام (غير مسجل)، فيمكننا استخدام مايلي:

```
pragma Shared(Variable_Name);
```

ويمكننا تطبيق هذه العملية على أي غرض سلمي. ومثال ذلك مايلي:

```
pragma Shared(Index);
```

ويجب استخدام هذه العملية، للسماح لعدة مهام بتغــير أو قــراءة القيمــة Index. وإنّ Shared تزود تزامن المهام، وهي ضرورية لضمان أنّ المعطيـات المشـتركة قـد تـمّ الرجوع إليها من قبل مهمتين أو أكثر، لأنّ المـترجم المحـدود، يمكـن أن يؤثـر علـى توضّع بعض الكيانات أثناء التنفيذ.

وبدلاً من استخدام متغيراتٍ عامة، يمكن استخدام مهمة لحجز أو تحريـر مـورد محدد. وعلى سبيل المثال، عند تغيير عنصر من قاعدة معطيات، فإننا نريد فقط لمهمة واحدة الوصول إلى سجل في نفس الوقت، كما يلي:

```
Seize_Resource;
--- Update the data here
Release_Resource;
```

ويعمـل المدخـلان Seize_Resource وRelease_Resource علـى شـكل أعـلام Semaphores. فإذا كان المورد مستخدماً من قبل مهمة، فإنه يجب علـى بقيـة المهـام الإنتظار عند المدخل Seize_Resource وفق ترتيب طلبهم، حتى المهمة التي في القسم الحرج تستدعي Release_Resource. ويمكننا تنفيذ الـ Semaphore، كما يلي:

```
task Semaphore is
 entry Seize_Resource;
 entry Release_Resource;
End Semaphore;
task body Semaphore is
 In_Use : Boolean := False;
Begin
 Loop
  select
   when not In_Use =>
    accept Seize_Resource;
    In_Use := True;
  or
   when In_Use =>
    accept Release_Resource;
```

```
      In_Use := False;
   End select;
 End Loop;
End Semaphore;
```

وإذا كان من الضروري حماية الوصول لأغراض معطيات عديدة، فإن أفضل طريقة لمشاركة المعطيات، تتمثل بخلق نوع مهمة، من أجل السيطرة على المنابع، كما يلي:

```
procedure Main is
 task type Resource is
   entry Seize;
   entry Release;
 end Resource;
```

وجسم نوع المهمة هذه، سيكون مكافئاً لجسم المهمة Semaphore. ويمكننا خلق سجل كما يلي، من أجل عناصر كهذه لحمايتها من المهام:

```
type Protected_Data is
 record
   Data_Item : Data;
   The_Resource : Resource;
 end record;
```

وبما أنّ نوع المهمة خاص ومحدود، فإنه لا يمكننا نسخ أو إسناد قيمة لأغراض من النوع Protected_Data. وبدلاً من ذلك، يمكننا تطبيق الخوارزمية التالية:

```
Shared_Data : Protected_Data;
task body Resource is separate;
Begin
 ...
  Shared_Data.The_Resource.Seize;
 -- Update Shared_Data.Data_Item here
   Shared_Data.The_Resource.Release;
 ...
End Main;
```

ويمكننا أن نبقى مجبرين للإتصال في بعض النقاط، باستخدام معطيـات عامـة ، مثـلاً، مـن أجـل أسـباب الفعاليـة. وهـي طريقـة لهـا نفس التـأثير كنـوع الإتصـالات المتزامنة ، الذي تمّ ذكره مسبقاً في هذا الفصل. وهذه الطريقة أساسـية ، وغـير بنيويـة. فمن السهل الوقوع بأخطاء في الخوارزمية. وأفضل طريقـة ، تتمثـل بتعليـب المعطيـات المشتركة في مهمة ، كما يلي:

```
task Protected_Item is
  entry Set(An_Item : in Item);
  entry Get(An_Item : out Item);
End Protected_Item;
task body Protected_Item is
  Local_Item : Item;
Begin
  Loop
    select
       accept Set(An_Item : in Item) do
         Local__Item := An_Item;
       End Set;
     or
       accept Get(An_Item : out Item) do
         An_Item := Local_Item;
        End Get;
    End select;
  End Loop;
End Protected_Item;
```

ففي هذه الحالة، يمكن لعدة مهام قراءة أو كتابة غرضٍ في نفس الوقت، ولكـن هذه المهمة تمنع كل وصول متزامن. وبالرغم من أننـا لـم نشـر إلـى الحـل هنـا، فإنـه يمكن تغيير هذه المهمة، للسماح لعدة قراء، ولعدة كتاب.

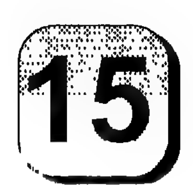
معالجة المقاطعات (Interrupts):

في النظم المحمولة، غالباً ما يجب علينا الإستجابة لحدث غير متزامن، يشار إليه بواسطة مقاطعة ذات بنية صلبة أو لينة. وفي معظم بقية لغات البرمجة، يجب أن نرتبط ببرنامج جزئي مكتوب بلغة التجميع، من أجل معالجة المقاطعة. بينما في لغة ADA، يمكننا معالجة المقاطعات باستدعاء مداخل مهمة. ولنفترض بأنّ الآلة المتاحة تسمح بمقاطعات شعاعية، وبالتالي، فإن مهمة ADA يمكنها الوصول للمقاطعة. وعلى سبيل المثال:

```
with System;
procedure Main is
 task Power_Failure is
  entry Fail;
  for Fail use at 16#1FE#;
 end Power_Failure;
 task body Power_Failure is
 Begin
  Loop
   accept Fail;
   -- do some actions
  End Loop;
 End Power_Failure;
Begin -- M
 ...
End Main;
```

وفي هذا المثال، فإننا قد ربطنا مقاطعة Power_Failure إلى البنية الصلبة، في الموضع ذي العنوان 1FE وفق الترميز الست عشري، بالرغم من إمكانية تواجد جسم المهمة في أي مكان في الذاكرة. وعندما نستقبل مقاطعة ذات بنية صلبة وينتقل تنفيذ الآلة إلى هذا الموضع، فإن العمل يكون مكافئاً لاستدعاء المدخل Fail. وعندها تستطيع المهمة قبول المدخل، وتتصرف كإجرائية لخدمة المقاطعة. وإن عبارة for، تكون مثالاً لتوصيف التمثيل الذي يسمح لنا باستخدام خدمات مرتبطة بالآلة. وسندرس هذه الوظائف بالتفصيل، في فصل لاحق.





معالجة الإستثناءات
Exception Handling

تصريح وإبراز الإستثناءات

معالجة الإستثناءات

تطبيق الإستثناءات



تعتبر الوثوقية عند التنفيذ، من العوامل الهامة في النظم المحمولة. فإذا كان لدينا نظام برمجي في قمر صناعي، أو في معمل طاقة ذرية، فلا يسمح لنا بأيّ خطأ. وعلى حال، يوجد في بعض الأحيان حالات إستثنائية خارجة عن سيطرتنا، مثل الأخطاء الناتجة عن البنية الصلبة للمحيطات، أو قراءة معطيات دخل غير متوقعة. ولايمكننا التنبؤ عن حدوث بعض الحوادث، ولكن يجب توقعها في النظام الموثوق.

وفي لغة ADA توجد عدة إستثناءات، حيث أن إسم الإستثناء يمثل حدثاً يسبب توقفاً في التنفيذ الطبيعي للبرنامج. وهذا الحدث، يمكن أن يكون خطأ ما، مثل Numeric_Error المعرّف مسبقاً، أو يمكن أن يكون شرطاً استثنائياً يتطلب معالجة خاصة، مثل طفحان ذاكرة مؤقتة (Buffer). وفي أفضل حالة، يجب أن يكون البرنامج قادراً على الإجابة على الإستثناءات. وإن بروز إستثناء يلفت انتباهنا على الشروط، لندعو الجواب بمعالجة الإستثناءات. وعندما نخطو في طريق التطوير غرضي التوجه، يمكن أن نحدد الشروط الإستثنائية المرتبطة بالخواص المنطقية للأغراض عالية المستوى، مثل الإستثناء Overflow خلال عملية Put المنفذة على رتل (Queue).

١٥ ـ ١ ـ تصريح وإبراز الإستثناءات

(Declaring and Raising Exceptions):

ويمكن لمبرمج في لغة ADA أن يصرح عن إستثناءات خاصة به، ولكنها تتضمن عدة شروط استثنائية معرّفة مسبقاً. وتوجد خمس إستثناءات معرّفة مسبقاً، تمّ التصريح عنها في الحزمة البرمجية Standard. وهذه الإستثناءات ما يلي: ,Constraint_Error Storage_Error ,Program_Error Numeric_Error وTasking_Error. وفي هذا الفصل، سنناقش الشروط التي من أجلها تبرز هذه الإستثناءات.

وتقنياً، فإن الإستثناء ليس غرضاً،و لكن يمكن أن يصرح المستخدم عن شروط، في الأماكن التي يمكن التصريح بها عن أغراض (ماعدا معاملات البرامج الجزئية، أو الوحدات المولّدة). والتصريح عن إستثناء معرّف من قبل المستخدم، يشبه تماماً

التصريح عن غرض. فهو يتألف من لائحة معرّفة، ونقطتين عموديتين، والكلمة المحجوزة exception ، كما توضحه الأمثلة التالية :

```
Above_Limits, Below_Limits  : exception;
Parity_Error                : exception;
Fatal_Disk_Error            : exception;
```

إن إسم الإستثناء المُعرّف من قبل المستخدم، يأخذ نفس مدى التصريح عن غرض، وأيضاً تأثير إستثناء، يمكن أن يمدّد لما وراء منظوره.

ويتم تنشيط الإستثناءات المعرّفة من قبل المستثمر صراحةً، باستخدام التعليمة raise، كما يلي:

```
raise Fatal_Disk_Error;
raise Above_Limits;
raise;
```

ففي المثال الأخير، إن (raise) يمكن أن يستخدم، فقط، في معالجة إستثناء، ويعيد إبراز الإستثناء الذي سبب تعليمة معالجة الإستثناء، ليصار إلى تنفيذه. وسنناقش وبتفصيل أكثر، معالجة الإستثناءات في ما بعد.

ويتم إبراز إستثناء معرّف مسبقاً (Predifined) ضمنياً أثناء زمن التنفيذ. وفيمايلي، بعض إستثناءات مختارة:

- Constraint_Error: ويتم إبراز هذا الإستثناء، عند حدوث خطأ في مجال، أو دليل، أو قيد أو مميز.
- Numeric_Error: ويتم إبراز هذا الإستثناء، عندما يكون ناتج عملية رقمية لا يمثل قيمة (مثل القسمة على صفر) أو تعطي قيمة خارج المجال المحدد.
- Pogram_Error : ويتم إبراز هذا الخطأ، في أحد الحالات التالية:
  - عدم إغلاق قسم Else، في حال تعليمة Select.
  - عند محاولة الوصول إلى برنامج جزئي، أو حزمة برمجية، أو مهمة قبل إنشائها.
  - عند محاولة الخروج من وظيفة، دون إرجاع قيمة.
  - عند اكتشاف خطأ في شرط .

- Storage_Error : ويتم إبراز هذا الإستثناء، عند امتلاء الذاكرة الديناميكية.
- Tasking_Error : ويتم إبراز هذا الإستثناء، أثناء اتصال المهمات الداخلية مع بعضها.

والإستثناء Constraint_Error، يبرز في عدة حالات، مثل تجاوز مجال غرض. والمثال التالي، يوضح ذلك:

```
procedure Out_Of_Range is
    Count : Natural := 0;
    Value  : Natural :=1;
Begin
    Count := Value - 4;-- the result = -3 which is not a value of  Natural.
End Out_Of_Range;
```

وإن الرجوع إلى كائن غير موجود في السياق الحالي، سيؤدي إلى إبراز الإستثناء Constraint_Error. والمثال التالي، يوضح ذلك:

```
with Text_IO; use Text_IO;
procedure Does_Not_Exist is
 type Text is access String;
 type Value (valid : Boolean := False) is
   record
     case Valid is
       when False => null;
       when True  => N : Integer;
     end case;
   end record;
package Value_IO is new Integer_IO(Integer);
use Value_IO;
Name  : Text;    -- Name has value null.
Result : Value;  -- Result.Valid is False.
Begin
 Put(Name.all);  -- Name.all does not exist, raise Constraint_Error
 Put(Result.N);  -- Result.N does not exist, raise Constraint_Error
End Does_Not_Exist;
```

توجد حالات غير واضح فيها فيما إذا سيستخدم الإستثناء Constraint_Error، أو الإستثناء Numeric_Error. ومثال ذلك ما يلي:

Too_Big : Integer := Integer'Last +1;

وفي بعض الحالات، فإن المبرمج المستخدم (defensive) يجب أن يكون جاهزاً لمعالجة هكذا إستثناءات.

وبما أنّه يمكن إبراز الإستثناء Constraint_Error بعدة طرق مختلفة، فيكون أكثر عملياً إذا تمّ إبرازه بشكل صريح للأخطاء التي تم تحديدها للمعالجة.

مثال: لنعتبر المكدس (Stack)، والذي يمثل كمصفوفة أحادية البعد، ودليل إلى قمة المكدس:

```
with Text_IO; use Text_IO;
procedure Explicit is
 subtype Item is Natural range 0..10;
 type List is array (Positive range <>) of Item;
 Max_Size : constant :=20;
 type Stack is
   record
     Items : List(1..Max_Size);
     Top : Natural :=0;
   end record;
  Overflow : exception;
  A_Stack : Stack;
  procedure Push (The_Item : in Item; OnTo : in out Stack) is separate;
Begin
 for Count in 1..25
   Loop
     Push(5,A_Stack);
   End Loop;
End Explicit;
```

وإن محاولة وضع عنصر على المكدس باستخدام Push مصيره الفشل، إذا كان في المكدس Max_Size عنصر قبل إستدعاء الإجرائية Push. وترميز الإجرائية Push، يتم على الشكل التالي (باكتشاف ضمني لامتلاء المكدس):

```
separate(Explicit)
procedure Push(The_Item : in Item; Onto : in out Stack)
Begin
  OnTo.Top := OnTo.Top + 1;
  OnTo.Items(OnTo.Top) := The_Item;
  exception
   when Constraint_Error => raise Overflow;
End Push;
```

وإن تعليمة الإسناد الثانية، ستؤدي لإبراز الإستثناء Constraint_Error، إذا كان Onto.Top أكبر تماماً من Max_Size. ولذلك، يتوجب على كاتب هذه الإجرائية، جعل هذا الشرط صريحاً، بهدف زيادة قابلية فهم و قابلية تغيير على هذه الإجرائية. وبالإضافة لذلك، عندما نغير جزئياً المكدس (مثلاً زيادة القمة Top) قبل فحص هذا الشرط، سنُرجع مكدساً شاذاً في حال إبراز الإستثناء Overflow، والسبب في ذلك، هو أنّ Top الحالي، أكبر تماماً من Max_Size . فمن الأفضل، وببساطة، فحص الشرط الذي يؤدي لإبراز الإستثناء Overflow في بداية الإجرائية، لتصبح على الشكل التالي:

```
Separate(Explicit)
procedure Push(The_Item : in Item; Onto : in out Stack) is
Begin
 if Onto.Top=Max_Size then
   raise Overflow;
   else
     Onto.Top:= Onto.Top+1;
     Onto.Items(Onto.Top):= The_Item;
 end if;
End Push;
```

ومثلما ذكرنا سابقاً، إن أحد الشروط الذي يسبب إبراز الإستثناء Program_Error، يتضمن اكتشاف شرط خاطئ (Erroneous Condition). ويعتبر الشرط خاطئاً، إذا لم يتم التنبؤ عن تأثيره، مثل، محاولة استخدام وظيفة لم تُعط قيمة لإرجاعها.

من المهم جداً ملاحظة أنّه عند إبراز إستثناء، فإنّ المعلومة الوحيدة الصالحة هي في الواقع حدوث الإستثناء. وبمعنى آخر، يشير الإستثناء فقط لوجود معضلة؛ وهو مسؤولية المبرمج لتطوير خوارزمية مطابقة للإجابة عن السبب.

وإن اكتشاف شروط إستثنائية، يؤدي لزيادة زمن التنفيذ. وعلى أي حال، من العملي استخدام إستثناءات، للشعور بزيادة فهم، ووثوقية، وصيانة النظام البرمجي. ويتم عادة الحصول على أرباح أعظم في التنفيذ، عند استخدام أفضل البنى العامة (Macrostructures)، بعكس، فعالية البنية الخاصة (Microstructures)، التي تعتمد على حذف الإستثناءات أثناء التنفيذ.

وعلى أي حال، إذا وجدت أسباب قوية وأساسية لإجراء ذلك، فإنّ لغة ADA تسمح بحذف مختلف فحوصات زمن التنفيذ (Run_Time). ويمكن أن نستخدم Pragma Suppress، في قسم التصريحات من وحدة أو كتلة برمجية. إذ أنّ أول معامل لـ Pragma يمثل معرّفاً، يمثل الفحص الذي سيُحذف، متبوعاً بمعامل ثان اختياري، يُعطي إسم نوع، إسم غرض، أو إسم وحدة لا نريد إجراء فحص زمن التنفيذ عليه. وإذا لم يُعط المعامل الثاني، فسيتم حذف فحص زمن التنفيذ، على الجزء الباقي من قسم التصريح. ويمكن استخدام إسماء الفحص التالية:

Suppression Of Constraint_Error *Checks*
Access_Check
Discriminant_Check
Index_Check
Length_Check
Range_Check
Suppression of Numeric_Error *Checks*
Division_Check

Overflow_Check
Suppression of Program_Error *Checks*
Elaboration_Check
Suppression of Storage_Error *Checks*
Storage_Check

فعلى سبيل المثال، إذا أردنا حذف فحص زمن التنفيذ، لقيود مجال لنوع يُدعى Index، يمكننا كتابة ما يلي:

```
Pragma Suppress(Range_Check, On => Index);
```

لاحظ كيفية تضمين المعامل الثاني، باستخدام الترميز الإسمي. مرة ثانية، لا ننصح باستخدام هذه الـ Pragma، دون أسباب مفروضة.

١٥ - ٢ - معالجة الإستثناءات (Handling Exceptions) :

عندما يحدث إستثناء، مثل القسمة على صفر، فإن معظم لغات البرمجة توقف المعالجة الطبيعية، ونظام الإستثمار يعاود التحكم . وفي أي نظام موثوق، لا نسمح لبرنامج بأن ينتهي بشكل غير نظامي؛ ويجب أن نمتلك طريقةً لإيقاف الإستثناء. ولتحقيق هذه الحاجة، تسمح لنا لغة ADA، بكتابة معالج الإستثناءات، لحجز الإستثناءات المعرّفة مسبقاً، وإستثناءات المستخدم. وعند بروز إستثناء في وحدة محددة، تتوقف معالجة تلك الوحدة، وينتقل التحكم لمعالج الإستثناء. ويظهر معالج الإستثناء بعد الكلمة المحجوزة exception ، والتي تمثل نهاية إختيارية لأي نافذة (وتمثل النافذة أي شيء له الشكل Begin Sequence_Of_Statements End;). فعلى سبيل المثال، تحدث النافذة في أي كتلة تعليمات، وفي جسم برنامج جزئي، وحزمة أو مهمة.

والقسم الإختياري من نافذة، له شكل مشابه تماماً لتعليمة Case. إذ أنّ كل عبارة When، تمثل معالج إستثناء، وتصمم الأجوبة الإستثناءات خاصة. ومثال ذلك، ما يلي:

```
with Text_IO; use Text_IO;
procedure Block_Example is
   procedure Open_Valve is
```

```
 Begin
  Put_Line("Open Valve");
 End Open_Valve;
 procedure Sound_Alarm is
 Begin
  Put_Line("Sound Alarm");
 End Sound_Alarm;
 procedure Close_Valve is
 Begin
  Put_Line("Close Valve");
 End Close_Value;
 procedure Log_Unknown_Error is
 Begin
  Put_Line("Log Unknown Error");
 End Log_Unknown_Error;
Begin
 -- any sequence of statements
  declare        -- the start of a block statement
   Low_Fluid_Level : exception;
  begin
   null;         -- any sequence of statements
   exception  -- marks the beginning of exception part
    when Low_Fluid_Level =>  -- one exception handler
      Open_Valve;
      Sound_Alarm;
    when Numeric_Error => --a second exception handler
      Close_Valve;
      raise;
    when others =>    -- the last exception handler
      Log_Unknown_Error;
  end;
 -- any sequence of statements
End Block_Example;
```

ففي هذا المثال، ولدعم حدوث الإستثناءات في سلسلة التعليمات (Sequence of statements) من أجل كتلة التعليمات المحددة، فإن المعالج يمكنه أن يسمي أي إستثناء مرئي (مثل Low_Fluid_Level، وأي إستثناء معرّف مسبقاً)، ومن ثمّ يحدد لائحة من التعليمات لتنفيذها، كجواب لكل إستثناء خاص. والعبارة Others، تعالج جميع الإستثناءات التي لم تحدد من قبل، أو جميع الإستثناءات التي لم تحدد إسماؤها في هذا المستوى من التصريح. وإذا تمّ إبراز الإستثناء Low_Fluid_Level ضمن كتلة التعليمات، فسيجيب معالج الإستثناءات على ذلك، بتنفيذ التعليمات Open_Value، ومن ثمّ Sound_Alarm. ويُعالج الإستثناء Numeric_Error، باستدعاء Close_Value، ومن ثمّ إعادة إبراز نفس الإستثناء. وتُعالج بقية الإستثناءات، باستدعاء Log_Unknown_Error.

ومن المهم ملاحظة أنّه بعد إنهاء تنفيذ معالج الإستثناء، فإنّ الشرط الإستثنائي لن يتواجد بعد ذلك، إذ يقال بأنّ الإستثناء قد تمّ تصغيره. وعندما ينتهي معالج الإستثناء من معالجاته، فإن التحكم لن يعود إلى النقطة التي برز منها الإستثناء، ولكن وببساطة، يتابع ما بعد نهاية النافذة التي عولجت فيها الإستثناءات. وفي القسم الثاني، سنفحص كيفية محاولة العملية ثانيةً. فإذا لم يبرز أي إستثناء، تستمر المعالجة بشكل طبيعي، وننتقل إلى أسفل النافذة، دون تنفيذ أيّ من تعليمات معالج الإستثناء.

في المناقشة السابقة، افترضنا بأنّ كل نافذة تحتوي على معالج إستثناء، لحجز جميع الإستثناءات المحلية. وعلى أي حال، لاتعتبر هذه الطريقة هي الأفضل من أجل تصميم البرنامج، وكبديل لذلك، في الأعمال البرمجية، من المفضل تطبيق مبدأ مستويات التجريد، فقط، مثلما أجرينا من أجل أنواع المعطيات. وكطريقةٍ مفضلة، يجب تصميم معالج إستثناءات، لحجز الإستثناءات كأدنى مستوى ممكن، حيث يتمكن البرنامج من الإجابة على الخطأ بطريقةٍ مناسبة. وغالباً، في مستويات منخفضة، تكون الإجابة عن إستثناءات معرّفة مسبقاً بسيطة، لإبراز إستثناءات المستخدم، وبهذه الطريقة، نحصل على إستثناء مطابق لمستوى التجريد.

ومثال ذلك، في حزمة برمجية لمعالجة المصفوفات، يمكن أن نصادف شرط القسمة على صفر، عند قلب مصفوفة؛ والإستثناء Numeric_Error، يمكن أن يبرز محلياً كجوابٍ للخطأ. ولا يمكن أن تجيب الحزمة البرمجية بشكل مطابق للشرط، حين لا تعرف غاية القلب. ولذلك، يمكننا تصدير الإستثناء Is_Singular، لاستدعاء إجرائية، تجيب على الإستثناء المحلي.

فإذا لم نُجب على إستثناء تم إبرازه في نافذة، فسينتشر الإستثناء، حتى يصل لمستوى يعالج الإستثناء. ومثال ذلك، ليكن لدينا البرنامج الرئيسي التالي:

```
with Text_IO; use Text_IO;
procedure Main is
 procedure Do_Somthing is
 Begin
  Put_Line("Do Somthing");
 End Do_Somthing;
 procedure Do_Somthing_Else is
 Begin
  Put_Line("Do Somthing Else");
 End Do_Somthing_Else;
 procedure Do_Somthing_More is
 Begin
  Put_Line("Do Somthing More");
 End Do_Somthing_More;
Begin
 -- any sequence of statements
 declare
  Local_Error : exception;
  Begin
   -- any sequence of statements
   exception
    when Local_Error =>
      Do_Somthing;
  End;
```

```
  -- any sequence of statements
  exception
    when Constraint_Error =>
      Do_Somthing_Else;
    when Numeric_Error =>
      Do_Somthing_More;
End Main;
```

وبما أنّ الإستثناءات المعرّفة مسبقاً، مصرح عنها في الحزمة البرمجية Standard، فإنّ هذه الإستثناءات مرئية بالنسبة للبرنامج الجزئي، وتعليمة الكتلة. وأيضاً، عرّفنا الإستثناء Local_Error، الذي يعتبر محلياً، إذ أنّه مرئي فقط لتعليمة الكتلة. وإذا تمّ إبراز الإستثناء Local_Error داخل الكتلة، فإن معالج الإستثناء المحلي سيحجز الخطأ، ويستدعي Do_Something. وبعد ذلك، سينتقل التحكم إلى نهاية الكتلة، بسبب أنّ تدفق التحكم تمّ تعريفه للإستثناءات.

ومن جهة أخرى، إذا تمّ إبراز الإستثناء Constraint_Error في داخل تعليمة الكتلة، فإنّ الإستثناء سينتشر إلى محتوى الجسم. وفي البرنامج الجزئي، يُحجز الإستثناء Constraint_Error المحلي، ومن ثمّ يُستدعى Do_Somthing_Else. وأيضاً، إذا تمّ إبراز الإستثناء Numeric_Error في داخل تعليمة الكتلة المحلية، فلا يوجد معالج إستثناء محلي؛ وبالتالي، سينتشر الإستثناء إلى الوحدة التي تحتوي معالجاً للإستثناء Numeric_Error .

وفي الحالة الأخيرة، إذا تمّ إبراز الإستثناء Program_Error في تعليمة الكتلة المحلية (أو في البرنامج الجزئي)، ربما خلال بناء قسم التصريح لتعليمة الكتلة، فلن نجد أي إستثناء على الإطلاق. وعندها، سينتقل التحكم إلى نظام الإستثمار. وبشكل عام، يمكننا حشر تعليمات كتل، مع أو بدون، معالج إستثناءات كل كتلة. وإذا برز إستثناء في تعليمة كتلة محددة، ولم يُعالج محلياً، فسينتشر إلى المستوى الذي يحتوي تعليمة الكتلة، إلى أن تتم معالجته.

وتحدث نفس قاعدة انتشار الإستثناءات للبرامج الجزئية، التي لا تمثل برنامجاً رئيسياً. وليكن المثال التالي:

```
with Text_IO; use Text_IO;
procedure Main is
 .......
  Type Small is digits 5 range 0.0..10.0;
   ...
  function Inverse(I : Float )
         return Small is
  begin
    return Small (1.0/I);
  exception
   when Numeric_Err0r =>
     return 10.0;
  end Inverse;
   ....
  Procedure Calling is
   X : Float := 0.0;
   Y : Small;
   Procedure Do_Something is
   Begin
    Put_Line(" do something"
Endive Do_Something;
  Begin
   ...
   Y := Inverse ( X) ;
   ...
   exception
    when Constraint_Error =>
      Do_Something;
   End Calling;
  Begin
 ....
  Calling ;
 ....
 End Main;
```

ففي هذا المثال، إذا استدعينا التابع الفرعي Inverse، مع معامل فعلي قيمته ٠,٠، سيبرز الإستثناء Numeric_Error في البرنامج الجزئي. وبما أنه يوجد معالج إستثناء محلي لحجز الخطأ، فإن المعالج سينفذ، ويعيد القيمة ٠,١٠. ومن جهة أخرى، إذا حاولنا الحصول على معكوس العدد٠,٠٠٠١. سيبرز الإستثناء Constraint_Error في داخل الوظيفة Inverse. ووفق قواعد اللغة، وبما أن معالج الإستثناء المحلي لا يمكن أن يحجز هذا الإستثناء، فإنّ نفس الإستثناء هذا، سيبرز في النقطة التي تُستدعى بها الإجرائية Calling. والإجرائية Calling تعالج الإستثناء Constraint_Error؛ وبالتالي، سيُصغر الإستثناء وستُنفذ التعليمة Do_Something.

لاحظ أيضاً، أنه إذا حدث إستثناء خلال بناء قسم التصريح لبرنامج جزئي، فسيبرز الإستثناء عند نقطة الإستدعاء. وإذا مثل البرنامج الجزئي برنامجاً رئيسياً، فسينتهي عندها تنفيذ الوحدة البرمجية. وبالفعل، فإن هذه الإستراتيجية مبررة، لأنه إذا حدثت أخطاء عند بناء كيانات قسم التصريح لبرنامج رئيسي، فإنه لا يمكننا التأكد من الحالة البدائية للبرنامج، وبالتالي، ستكون المعالجة غير موثوقة.

والآن، سنعتبر قواعد معالجة الإستثناءات للوحدات البرمجية، التي على شكل حزم برمجية

ومهمات. فإذا أُعطيت حزمة برمجية لا تشكل وحدة مكتبية، وإذا حدث إستثناء غير معالج في سلسلة تعليمات جسم الحزمة البرمجية، أو أثناء بناء قسم تصريح تلك الحزمة البرمجية، فإنّ الإستثناء سينتشر إلى نقطة تتبع مباشرةً جسم الحزمة البرمجية، في الوحدة التي تحتوي تصريح الحزمة. ومن جهة أخرى، إذا كانت الحزمة البرمجية تمثل وحدة مكتبية، وحدث إستثناء، فسينتهي تنفيذ البرنامج الرئيسي، لنفس السبب المذكور في الفقرة السابقة. وإذا برز إستثناء عند بناء قسم التصريح لمهمة، فإنّ الإستثناء Tasking_Error، سوف يبرز في نقطة تنشط المهمة، وستُؤشّر المهمة باكتمالها. وإذا برز إستثناء غير معالج خلال تنفيذ جسم مهمة، فإنّ هذه المهمة تصبح مكتملةً، ولن ينتشر الإستثناء بعد ذلك.

وأيضاً، يجب أن نعالج حالةً خاصةً من حدوث الإستثناء، خلال تنفيذ الموعد بين المهمات. وبسبب عدم تناظر آلية المهمات في لغة ADA، فإنّ قواعد لغة ADA تحدد بأنّ الإستثناء Tasking_Error، سيبرز عند إستدعاء مهمة في مكان إستدعاء المدخل، إذا كانت المهمة المستدعاة غير نشطة عند استدعاء المدخل، أو إذا اكتملت قبل قبول إستدعاء المدخل.

وإذا انتهت المهمة المستدعاة بشكل غير نظامي خلال موعد، أيضاً، سيبرز الإستثناء Tasking_Error في المهمة المستدعية. وإذا توقفت المهمة المستدعية خلال موعد، فلا ينتشر أي إستثناء إلى المهمة المستدعاة. وأخيراً، إذا برز إستثناء خلال تعليمة accept، فإننا نترك تنفيذ تعليمة accept، وسيتم إبراز نفس الإستثناء داخل المهمة التي تحتوي تعليمة accept. وأيضاً، يبرز الإستثناء في المهمة المستدعية، في نقطة إستدعاء المدخل.

وقبل إنهاء هذا الفصل، نحتاج للرجوع إلى فكرة إنتشار إستثناء، إلى ما بعد مداه. وبمعنى آخر، إن الإستثناء المعرّف إسمه محلياً، يمكن أن ينتشر بعد الوحدة المصرح عنه بها، إلى نقطة لا يوجد فيها إسمٌ للإستثناء، يمكننا الرجوع إليه. ولنعتبر الحالة التالية:

```
with Text_IO; use Text_IO;
procedure Local is
 procedure Do_Something is
  Begin
    Put_Line("Do_Something");
  End Do_Something;
 procedure Put_Line("Do_Something Else");
```

إنّ الإستثناء Local_Exception، يعرّف فقط داخل الكتلة المحلية (Local Block).

فإذا أبرزنا الإستثناء Local_Exception، فإنّه سينتشر إلى خارج الكتلة، لأننا لم نوفر معالج إستثناء محلي. وفي الكتلة الخارجية (Outer Block)، لا نستطيع معرفة إسم هذا الإستثناء، لأننا خارج مدى تعليمة الكتلة، الذي تم التصريح فيها عن الإستثناء Local_Exception. ويمكننا التقاط هذا الإستثناء، فقط، باستخدام عبارة others.

```
Begin
 ...
  declare                                        ┐ Outer
   ...                                           │ Block
  Begin                                          │
   ...                                           │
   declare                        ┐ Local        │
    Local_Exception : exception;  │ Block        │
   Begin                          │              │
    ...                           │              │
    raise Local_Exception;        │              │
    ...                           │              │
   End;                           ┘              │
  ...                                            │
  exception                                      │
   when Numeric_Error =>                         │
    Do_Something;                                │
   when others =>                                │
    Do_Something_Else;                           │
  End;                                           ┘
  ...
 End Local
```

١٥ ـ ٣ ـ تطبيق الإستثناءات (Applying Exceptions):

كما شاهدنا ومن أجل كل بنية من بنى لغة ADA التي تمّت دراستها حتى الآن، فإنه توجد طرق جيدة وطرق سيئة لتطبيق الإستثناءات. على سبيل المثال، يجب ألا نستخدم الإستثناءات للحصول على نوع من تسهيلات Goto الضمنية. فيما بعد وعند نمذجة الحلول، يجب علينا محاولة تحديد شروط الأخطاء الممكنة لأغراضنا ولخوارزميتنا، وألا نستخدم صراحة الإستثناءات إلا من أجل التخطيط لمعالجته.

عند إبراز إستثناء، توجد عدة أنواع من الأعمال الممكنة، هذه الأعمال مايلي:

- ترك تنفيذ الوحدة (Abandon The Execution Of The Unit) .
- إعادة محاولة العملية مرة ثانية (Try The Operation Again) .
- استخدام طريقة مختلفة (Use An Alternative Approach) .
- تصليح سبب الخطأ (Repair The Cause Of Error) .

عادة لا ينصح باستخدام العمل الأول (ترك تنفيذ الوحـدة)؛ إذا حـدث خطـأ، يجب أن نجيب بعمل معيـن. تـرك التنفيـذ يكـون مناسباً ، إذا كـان مـن المسـتحيل المتابعة في معالجة الوحدة الحالية أو تعليمة الكتلة. على سبيل المثال إذا كان هنـالك خطأ قاتل في جهاز طرفي ليمنعنا من متابعة عمليات الدخل/الخرج؛ فإن عملنا يكـون إيقاف المعالجة واسترجاع هذا الشرط.

كما يشير المثال التالي، يمكننا تصدير إستثناءات على شكل جزء من توصيف حزمة برمجية :

```
package IO_Interface is
  procedure Put(A_Character : in Character);
 Timeout : exception;
End IO_Interface;
package body IO_Interface is
  Milliseconds : constant Duration :=0.01;
  task IO_Driver is
    entry Send(C : in Character);
  End IO_Driver;
  task body IO_Driver is
  Begin
    Loop
        accept Send(C : in Character) do
      -- peripheral dependent code
    End Send;
     ...
    End Loop;
  End IO_Driver;
  procedure Put(A_Character : in Character) is
   Begin
     select
       IO_Driver.Send(C);
      or
       delay 5*Milliseconds;
       raise Timeout;
     End select;
   End Put;
 End IO_Interface;
```

في هذه الحالة، لقد عرفنا IO_Interface التي تضع في رتل انتظار طلبات وضع قيم المحارف وفق Put؛ في جسم الحزمة البرمجية توجد مهمة تنفذ الاتصال المخزّن. عندما نجري Put على قيمة، عندها نستدعي وبشكل غير مباشر المهمة من خلال برنامج جزئي آخر. إذا لم يجب الطرفي خلال الفترة الزمنية المحددة، سنفترض وجود عطل في الطرفي. في خوارزميتنا، لقد اخترنا إبراز الإستثناء Timeout، الذي سيصدر إلى النقطة التي تمّ فيها استدعاء البرنامج الجزئي Put . بما أنّه قمنا بتسمية الإستثناء Timeout في قسم توصيف الحزمة البرمجية، يمكن لمستخدمي هذه الحزمة البرمجية كتابة معالج إستثناء ليعود لهذا الإستثناء.

بدلاً من ترك التنفيذ، يمكننا اختيار العمل الثاني (إعادة محاولة العملية مرة ثانية) من أجل تكرار العملية بعد إبراز الإستثناء. التقنية التي نستخدمها لتكرار العملية تتمثل بالتصريح عن تعليمة كتلة محلية تعلب الخوارزمية التي نريد حمايتها، ومن ثمّ وضع الكتلة داخل حلقة تكرر العملية. على سبيل المثال، الترميز التالي سيعطى قيمة مرقمة مُدخلة :

```
with Text_IO; use Text_IO;
procedure Second_Alternative_1 is
  type Response is (Up, Down, Left, Right);
  package Response_IO is new Enumeration_IO(Response);
  use Response_IO;
  User_Request : Response;
Begin
  Loop              -- repeat the operation
    Begin           -- start of the defended code
      Put(">");
      Get(User_Request);
      Skip_Line;
      exit;         -- exit if there is no exception
      exception
        when Data_Error =>
          Skip_Line;          -- skip the bad input
          Put_Line("Invalide response; enter only Up, Down, Left, or Right .");
    End;
  End Loop;
End Second_Alternative_1;
```

عند دخولنا الكتلة، نحصل على الإشارة > بعد ذلك ننتظر دخل من User_Request. إذا كان الدخل صالحاً، ننتقل للتعليمة التالية ونخرج من الحلقة. إذا أدخل المستخدم شيئاً لا ينتمي للقيم (Up, Down, Left, Right)، سيبرز الإستثناء Data_Error بواسطة حزمة الدخل/الخرج. بعد كتابة رسالة الخطأ، يتم ترك تنفيذ الكتلة. على أي حال، بما أنّ الكتلة داخل الحلقة، نعود مرة ثانية لبداية الحلقة وندخل الكتلة من جديد، مكررين المعالجة حتى يُدخل المستثمر جواباً صالحاً يجعلنا نترك الحلقة.

إن الدخول بحلقة حتى يتم الحصول على إجابة صالحة للمستخدم صالح، هو بالتأكيد قرار تصميم مبرر. على أي حال، توجد بعض الحالات نرغب من خلالها بإعادة المحاولة لعملية عدد محدود من المرات فقط – كما لو أننا حاولنا الاتصال بجهاز طرفي. كمثال على ذلك، لنأخذ إجرائية الدخل الأصلية ولنجر تغيراً عليها بحيث نحاول خمس مرات فقط:

```
with Text_IO; use Text_IO;
procedure Second_Alternative_2 is
 type Response is (Up, Down, Left, Right);
 package Response_IO is new Enumeration_IO(Response);
 use Response_IO;
 User_Request : Response;
Begin
 for Repeat_Count in 1..5   -- repeat the operation
 Loop
  Begin         -- start of the defended code
   Put(">");
   Get(User_Request);
   Skip_Line;
   exit;        -- exit if there is no exception
   exception
    when Data_Error =>
     Skip_Line;
    if Repeat_Count < 5 then
     Put_Line("Invalide response;" &
```

```
        "enter only Up, Down, Left, or Right .");
      else
    Put_Line("You tried too many times.Up is assumed.");
        User_Request := Up;
      End if;
  End;
 End Loop;
End Second_Alternative_2;
```

في هذه الحالة، إذا تمّ إبراز الإستثناء Data_Error، فإنّ معالج الإستثناء يحدد كم هو عدد المحاولات التي قمنا بها، ويطبع الرسالة الموافقة. بدلاً من تصحيح خطأ المستخدم، كان بإمكاننا إبراز إستثناء يمرر الشرط إلى وحدة البرنامج الأعلى مباشرة.

كجواب آخر ممكن للإستثناء أن يتم باستخدام طريقة مختلفة. (الحالة الثالثة) كمثال على ذلك، في نظام حرج لتدوير الرسائل، والذي هو على مستوى عال من الوثوقية؛ نصمم هنا عدداً من مهام الاتصالات الزائدة. وهكذا، إذا حاولنا دخول مهمة محددة واستقبال الإستثناء Tasking_Error الذي يشير إلى إخفاق في ممر الاتصال، عندئذ نختار طريقاً مختلفاً. لنعتبر المثال التالي:

```
Begin
 Send_message_To_Path_1(Critical_Message);
 exception
  when Tasking_error => Send_Message_To_Path_2(Critical_Message);
End;
```

كما سبق، لاحظ كيف استخدمنا تعليمة كتلة مع معالج إستثناء محلي لتعريف مقطع من ترميز.

إذا طبقنا مجموعة من المهام بدلاً من استخدام مهام اتصالات شخصية، يمكننا دمج هذه الطريقة المختلفة مع إعادة المحاولة مرة ثانية (الطريقة السابقة) ، كما هو واضح في المثال التالي:

```
with Text_IO; use Text_IO;
procedure task_Error is
  task type Message_Task is
```

```
   entry Put(Message : String);
  End Message_Task;
  Send_Message : array (1..10) of Message_Task;
  task body Message_Task is
  Begin
   Loop
    accept Put(Message : String) do
     Text_IO.Put_Line(Message);
    End Put;
   End Loop;
  End Message_Task;
Begin   -- Task_Error
  for Index in Send_Message'range
  Loop
   Begin
    Send_Message(Index).Put("Critical_Message");
    exit;
    exception
     when Tasking_Error =>
      if Index = Send_Message_Last then
       Send_Alert_To_Operator;
       raise;  -- propagate Tasking_Error
      End if;
   End;
  End Loop;
End Task_Error;
```

في هذا المثال، حاولنا إرسال رسالة إلى واحد من عشر مهام رسائل ممكنة. إذا أخفقنا في تمرير الرسالة عشر مرات (على سبيل المثال، إذا انتهت جميع المهام المستدعاة)، عندها ننفذSend_Alert_To_Operator . بالإضافة لذلك، نبرز نفس الإستثناء (Tasking_Error) ليتم نشره للمعالج التالي.

التطبيق الرابع لإستثناءات لغة ADA يتمثل بتصحيح سبب الخطأ. من أجل معالج إستثناء في برنامج جزئي، جميع الأغراض المحلية للبرنامج الجزئي، بما في ذلك المعاملات الصورية، تكون مرئية لنا. يمكننا استخدام هذه الإغراض لتصحيح

سبب الخطأ. على سبيل المثال، في نظام تحكم، يمكننا استدعاء برنامج جزئي لقيادة حركة جنيح. ستكون هناك آلية تخديم رجعية تربط تأثير هذا الطلب: إذا أرسلنا طلباً هاماً جداً، يمكننا تجاوز حدود حركة الجنيح. عندها يجب إبراز إستثناء قبل تدمير الجنيح. يمكن الحصول على ذلك وفق الترميز التالي:

```
with Text_IO; use Text_IO;
procedure Main is
 procedure Move_Rudder(Amount : in Integer) is
  Rudder_Stressed : exception;
 Begin
  -- send Command to rudder servomechanism,
  -- the local exception may be raised here
  if Amount > 0 then
    raise Rudder_Stressed;
   else
    return;
  End if;
  exception
   when Rudder_Stressed =>
    Put_Line("reduce amount");
    Move_Rudder (Amount / 2 );   -- retry
 End Move_Rudder;
Begin  -- Main
 Move_Rudder(10);
End Main;
```





16

تمثيلات الآلة
Machine Representation

توصيفات التمثيل

الميزات المرتبطة بالنظام

التحويل غير المضبوط



بما أنّ لغات التجميع تجبرنا بالعمل بمستويات الآلة الأساسية، فإن لغات البرمجة عالية المستوى، تجبرنا عادةً على التعامل فقط مع مستويات أكثر تجريداً. وبما أنّ البرمجة بلغة عالية المستوى، أكثر إنتاجية من البرمجة بلغة التجميع، فلا تعتبر هذه مشكلة. وعلى أي حال، يجب علينا الرجوع في بعض الأحيان للميزات المرتبطة بالنظام، مثل موضع بوابة دخل/خرج، أو تمثيل بعض بنى المعطيات في الذاكرة. وفي الماضي، وبسبب عدم إمكانية تزويد لغات البرمجة بقدرة تعبير مناسبة، كنا مجبرين لاستخدام برامج مكتوبة بلغة عالية المستوى، مع لغة التجميع في الحلول، وهي طريقة تعقّد الحلول، وتعيق قراءة، وصيانة البرامج.

وبشكل مثالي، فإننا نرغب بلغة تزودنا بأساليب للتعبير عن الميزات منخفضة المستوى للآلة، بطريقة عالية المستوى. وتزود لغة ADA تطورات برمجية بكلا المستويين. وفي هذا الفصل، سندرس بنى اللغة من أجل البرمجة المتعلقة بالنظام.

١٦ ـ ١ ـ توصيفات التمثيل (Representation Specifications):

إن توصيفات التمثيل، تصف كيفية بناء كيانات الحلول على الآلة المتاحة. وإن أيّاً من التوصيفات، يمكن أن يظهر في قسم تصريح وحدة برمجية، أو في قسم توصيف المهام، والحزم البرمجية، في لغة ADA. ومن المهم ملاحظة بأنّ كل نوع معطيات، يمكن أن يملك واحداً، وواحداً فقط، من التمثيل. وأكثر من ذلك، يمكننا بناء توصيفات تمثيل، فقط، بعد تصريحنا عن النوع، وقبل التصريح عن أي غرض من النوع، أو استخدام الكيان (كتعبير).

ويجب تطبيق توصيفات التمثيل فقط :

- من أجل أهداف الفعالية.
- للسماح لنا بالرجوع إلى ميزات منخفضة المستوى، مع إسماء ADA العادية.
- عند طلب التخاطب، مع نظم خارجية، أو موجودة.

وعندما نكون بحاجة لتطبيق هذه الميزات منخفضة المستوى، تسمح لنا لغة ADA بخلق تجريدات حولها، في حدود عالية المستوى.

ويمكننا صراحةً، جدولة معيار عام، من أجل تمثيل بنى المعطيات، ومن أجل جعل الترميز أمثلياً، باستخدام عمليين Pragmas، معرفين مسبقاً، كما يلي:

```
pragma Pack(Some_type);
pragma Optimize(Time);
pragma Optimize(Space);
```

وفي الحالة الأولى، فإن العملي Pack، يشير بأنّ الأغراض التي لها النوع Some_type (على سبيل المثال، مصفوفة من عناصر منطقية)، سيتم ضغطها لحذف أي مجال، وفق الطريقة التي تم تخزينها بها. بينما في الحالتين الأخيرتين، فيتم إعطاء توجيه للمترجم، معتبرين معيار الأمثلية الأولي، من أجل ترجمة محددة. وهذه العمليات، وما تبقى منها، سنشرحها بالتفصيل فيما بعد.

وتزود لغة ADA بأربع عبارات من أجل تمثيلات مخصصة. وهذه العبارات، هي مايلي:

- الطول Length .
- المرقم Enumeration .
- التسجيلة Record .
- العنوان Address .

وإن عبارة الطول، تتحكم بكمية الذاكرة المرتبطة بكيان محدد، ولها الشكل التالي:

```
for attribute use simple_expression;
```

وهنا يشير *attribute*، إلى نوع توصيف الطول، والتعبير *simple_expression*، يعطي قيمةً رقمية، بحيث يرتبط معناها بـ *attribute*. والواصفات المطابقة بتوصيف التمثيل هذا، هما Size, Storage_Size, & Small. وكل واحدة من هذه الواصفات، سيتم شرحها بالتفصيل، في فصلٍ لاحق. وبشكل خاص، يسمح لنا استخدام الواصف

Size، بتعين حد أقصى لعدد الخانات الثنائية، المحجـوزة لأغـراض مـن نـوع معيـن. ومثال ذلك ما يلي:

```
Bits : constant :=1;
type My_Integer is range -100..100;
for My_Integer'Size use 8*Bits;
```

لاحظ تصريح الثابت Bits ، إذ تمّ التصريح عنه لتسهيل القراءة. وكنتيجة لهذا التصريح، فإن كل غرض من النوع My_Integer، لـن يحجـز أكـثر مـن ثمـان خانـات ثنائية. ويجب على المستخدم لتوصيفٍ مـا، أن يتحقـق مـن أنّ تعييـن حجـم مصغـر لنوع، سيؤدي لتأثيرات مختلفة على ترميز الولوج acces code، والذي مـن الممكـن أن يؤثر على سرعة التنفيذ. ويمكن الحصول على عبارة طول غـير صالحـة، مثـلاً، تعييـن أربع خانات ثنائية للنوع My_Integer؛ إذ أنّ القيمة ٥٠، لا يمكن تمثيلها على أربـع خانات ثنائية.

ويمكننا أيضاً، تعيين كمية الذاكرة المسموحة لمجموعة من الأغراض المحجوزة، أو من أجل تنشيط غرض مهمة. وعلى سبيل المثال:

```
procedure Main is
 Bits : Constant := 1;
 type My_Integer is range -100..100;
 for My_Integer'Size use 8*Bits;
 Bytes : constant :=8*Bits;
 Kilo_Bytes : constant : 1024*Bytes;
 type Buffer;
 type Record_Pointer is access Buffer;
 for Record_Pointer'Storage_Size use 100*Bytes;
 task type Watchdog_Task is
   ...
 End Watchdog_Task;
 for Watchdog_Task'Storage_Size use 3*Kilo_Bytes;
 type Buffer is
  B : String(1..100);
 End record;
 task body Watchdog_Task is
```

```
 Begin
   ...
 End Watchdog_Task;
Begin -- Main
 ...
End Main;
```

ففي حالة Record_Pointer، إن توصيف التمثيل يحجز 100 Bytes من الذاكرة، من أجل جميع الأغراض المصممة بواسطة قيم الوصول لـ Record_Pointer. ومن أجل المهمة، يعود Storage_Size لكمية الذاكرة المحجوزة من أجل تنشيط المهمة المعطية. (يتضمن ذلك المعطيات بدون الترمين)

لاحظ بأنّ الواصف Storage_Size، قد تمّ تعريفه بعدد من وحدات الذاكرة المحجوزة، إذ أنّ وحدات الذاكرة عادةً، تشير لطول كلمة الآلة. وتعرف الحزمة البرمجية System، المعرّفة مسبقاً ثابتاً يسمى Storage_Unit، ويشير هذا الثابت لعدد الخانات الثنائية، لكل وحدة ذاكرة. (إن توصيفات الحزمة البرمجية System، ستُحدد في فصل لاحق). لاحظ بأنّه يمكن لـ System.Storage_Unit، أن يختلف من آلة لأخرى. وفي حالة الواصف Storage_Size، يبرز الإستثناء Storage_Error، إذا تمّ تجاوز الذاكرة المحجوزة.

والإستخدام الأخير لعبارة الطول، يتمثل بتحديد الدلتا (الممين الفعلية لنوع الأرقام الممثلة بالفاصلة الثابتة، كما هو مبين فيما يلي:

```
type Radians is delta 0.001 range 0.0..1.0;
for Radians'Small use 0.001;
```

ففي هذه الحالة، إن قيمة Small المعطية في عبارة الطول، يجب أن تكون أصغر أو تساوي قيمة الدلتا (الممين)، المحددة عند التصريح عن النوع. وتستخدم هذه البنية، من أجل زيادة دقة النوع الممثل وفق الفاصلة الثابتة.

والحالة الثانية من توصيفات التمثيل، من أجل الأنواع الترقيمية. ويسمح هذا التوصيف، بالإشارة للترميزات الداخلية للقيم من ذلك النوع. وهكذا توصيف، يشبه

توصيف الطول، لكنه يستخدم كلياً من أجل تعيين التطابق، بدءاً من قيم النوع الترقيمي إلى الترميزات الداخلية. فعلى سبيل المثال:

```
type Response is (Up, Down, Left, Right);
for Response use (Up => 1,
                  Down => 2,
                  Left => 4,
                  Right => 8);
```

فوفق هذا المثال، أجرينا تطابقاً للأحرف Up, Down, Left, Right، وذلك بإعطائها ترميزات داخلية وحيدة (القيم ١، ٢، ٤، ٨). وبالإضافة لذلك، لا يمكن للترميزات الداخلية أن تكون متقاربة (كما هو الحال في المثال السابق). ومهما كان التمثيل الداخلي، فإنّ العلاقات Up< Down< Left < Right تبقى صحيحة، ولا تؤثر على استخدام الواصفات Pos, Pred, Succ. فعلى سبيل المثال، إنّ قيمة Pos لـ Up, Down, Left, Right، تأخذ بالترتيب القيم٠، ١، ٢، ٣. ويتم إعتماد الترميزات الداخلية، ليتم استخدامها فقط عندما يتم تمرير القيم الترقيمية خارج ADA، للبنية الصلبة، أو لترميز مكتوب بلغة برمجة أخرى. وفي الواقع، إن الحزمة البرمجية Standard، تعين الترميزات للنوع المعرّف مسبقاً Character، ليتكيف مع الترميز المعياري ASCII.

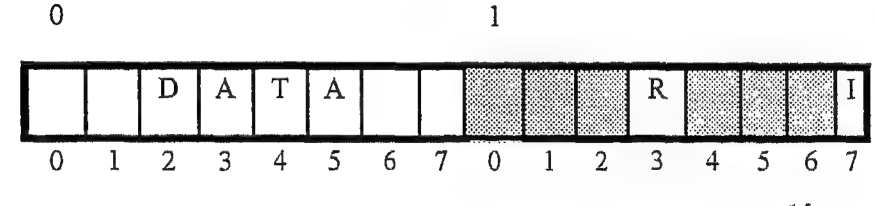
وتشمل الحالة الثالثة، تمثيل أنواع المعطيات المسجّلة. وبشكل خاص، يمكننا وصف تنظيم التسجيلة الطبيعية بدلالة وحدات الذاكرة، بالإضافة لموضع كل مكون تسجيلة، ضمن وحدات الذاكرة تلك. والإستخدام النموذجي لتلك البنية، يتمثل بوصف البنى المادية القابلة للعنونة لآلتنا في أعلى مستوى. فعلى سبيل المثال، فإن تجريدنا لبوابة ذاكرة مع كلمة حالتها، يمكن وصفه كما يلي:

```
type IO_Port is
  record
    Data   : integer range 0..255;
    Ready, Interrupt_Enabled : Boolean;
  End record;
```

وبفرض أن كل وحدة تخزين في آلتنا تساوي Byte 1، كعرض وأنّ أغراض IO_Port تتمثل بـ 2 Bytes متجاورين من الذاكرة. فيمكن كتابة تمثيل التسجيلة، كما يلي:

```
for IO_Port use
 record at mod 2;
  Data             at 0 range 0..7;
  Ready            at 1 range 3..3;
  Interrupts_Enabled at 1 range 7..7;
 End record;
```

ففي هذا النوع من توصيف التمثيل، يمكننا استخدام البنى At, mod لتعيين تنظيم التسجيلة، بدلالة جميع وحدات الذاكرة الضرورية. وهنا، ستملك الأغراض IO_Portعناوين بداية، هي من مضاعفات الـ ٢، وتعين العبارة at الموضع النسبي لمكوّن، معبر عنه بواحدات الذاكرة. ويشير المثال السابق، بأنّ المكون Ready من الأغراض IO_Port، تتوضع في البايت الثاني من الغرض. وأخيراً، فإن البنية range، تعين الخانات الثنائية الحالية داخل البايت، المستخدمة من قبل المكوّن. (لاحظ بأنّ الخانة الثنائية الأولى، تترقم بصفر). وفي مثالنا، إن المكون Interrupts_Enabled يتوضع في الخانة الثنائية رقم ٧ من البايت الثاني، ويحجز فقط خانة ثنائية واحدة. وهكذا توضع، يمكن وصفه كما يلي، بحيث أن المناطق المظللة تمثل خانات ثنائية غير مستخدمة:



إذ أنّ R=Ready , I=Interrupts_Enabled

والنوع الأخير من التمثيل الذي تقدمه ADA، يتمثل بتوصيف العناوين. وبالتالي، تأخذ البنية شكلاً مشابهاً لبقية التوصيفات. فعلى سبيل المثال، يمكن توصيف موضع ثابت أو متغير، كما يلي:

```
D_To_A_Convereter : IO_Port;
For D_To_A_Converter use at 16#177F6#;
```

ففي هذه الحالة، قد صرحنا عن غرض من النوع IO_Port، وعيناه بأن يتوضع بالعنوان المطلق 16##. والقيمة المعطية بعد use at، يجب أن تكون قيمةً من النوع System.Address، وهو نوع يتعلق بالنظام. وللتبسيط، فإن أمثلة عبارات العنونة في هذا الفصل، تفترض بأنّ النوع System.Address هو نوع صحيح. وبالحقيقية، لا يحتاج System.Address بأن يكون من النوع الصحيح؛ ويمكن أن يكون أي نوع، بحيث يختاره مصمم النظام لتمثيل مخطط العنونة لجزء خاص من البنية الصلبة. ومن أجل بعض النظم، يمكن أن تحتوي تسجيلة رقم عنوان القطاع وإنزياحه ،كما يلي:

```
type Address is
  record
   Segment : Word;
   Offset    : Offset_Type;
  End record;
```

أيضاً، يمكن استخدام عبارات العنونة، للإشارة لعنوان البداية لوحدة برنامج كبرنامج جزئي، أو حزمةٍ برمجية، أو مهمة. والعنوان المخصص، يشير إلى بداية ترميز الآلة المرتبط بجسم الوحدة البرمجية. فعلى سبيل المثال، يمكن لنظام استثمار آلتنا، أن يملك إجرائيةً تتوضع في العنوان المطلق 8#76#، الذي ينفذ توقفاً بطيئاً للنظام. ومن أجل الرجوع إلى هذه الإجرائية بدلالة ADA العادية، نصرح ما يلي:

```
procedure Power_Down;
for Power_Down use at 8#76#;
```

ووفق هذا، وعندما نستدعي Power_Down من برامج لغة ADA، نستدعي بالفعل، إجرائية مكتوبةً بلغة الآلة. لاحظ بأنّه يجب عدم استخدام هذا، من أجل وضع وحدات برمجية في ذاكرة غير مسموح بها.

والتطبيق الأخير لعبارة العنونة، يسمح بربط مدخل مهمة بمقاطعة، مثلما لاحظناه في الفصل الخاص بالمهام. كما يلي:

```
task Air_Conditioner_Failure is
  entry Temperature_Interrupt;
```

```
 for Temperatere_Interrupt use at 16#3E#;
End Air_Conditioner_Failure;
task body Air_Conditioner_Failure is
Begin
 Loop
 accept Temperature_Interrupt do
  Power_Down;
 End Temperature_Interrupt;
 End Loop;
End Air_Conditioner_Failure;
```

ففي هذا المثال، إذا كان لدينا مقاطعة صلبة في الموضع 3E (وفق الترميز الست عشري)، ستلتقط المهمة Air_Conditioner_Failure الإشارة، وتستدعي Power_Down. لاحظ بأنّه لا يمكننا ربط أكثر من مدخل مهمة واحدة، لكل مقاطعة محددة؛ فإذا قمنا بذلك، عندها يعتبر البرنامج خاطئاً.

وحتى الآن، لقد أكدنا بأنّ ADA، تسمح بتمثيل واحد فقط لكل نوع. وعلى أي حال، يمكن أن توجد حالات تكون أكثر فاعليةً بوجود تمثيلين، أو ربما توافق بشكل أفضل نظرتنا للعالم. فعلى سبيل المثال، إننا نحتاج لتنفيذ مجموعة ضخمة من تسجيلات أنظمة القياس (telemetry). وعندما نحفظ تسجيلةً على قرص، نحتاج لضغط المعطيات، وحفظها في مكان آمن. وإجراء حسابات على معطيات مضغوطة يكون غير فعال، بينما نحن نجري بشكل مستمر إلغاء حفظ وإعادة حفظ المعطيات. ويمكن لحلّنا أن يملك تمثيلين. التمثيل الأول، من أجل الحسابات. والتمثيل الثاني، من أجل التخزين. وهذا يبيح للمبرمج التحكم صراحةً بضغط المعطيات.

فعلى سبيل المثال، يمكننا التصريح عن التسجيلة Telemetry دون توصيف تمثيل، سامحين للمترجم باختيار تمثيلها الأمثلي. وعندها، سنصرح عن تمثيل مضغوط من نفس التسجيلة:

```
type Telemetry is
 record
  ...
 End record;
```

```
type Compressed_Telemetry is new Telemetry;
pragma Pack (Compressed_Telemetry);
```

ويوجد حالياً، نوعان مرتبطان بتمثيلين مختلفين. (يمكننا كتابة برنامج فحص لتقييم واصفات الحجوم لهذه الأنواع، لتعيين، فقط، وكم من الذاكرة حفظنا باستخدام Pack). كما لاحظنا في فصل سابق، يمكن أن نُجري تحويلاً بين النوع المشتق وأصله، كما يلي:

```
subtype Unpack is Telemetry;
subtype Pack is Compressed_Telemetry;
Computational_Data : Telemetry;
Storage_Data : Compressed_Telemetry;
Begin
  Computational_Data := Unpack(Storage_Data);
  Storage_Data          := Pack(Comutational_Data);
End Main;
```

لاحظ بأنّ استخدام الأنواع الجزئية، يجعل محولات الأنواع مقروءةً بصورة أفضل.

١٦ - ٢ - الميزات المرتبطة بالنظام

(System – Dependent Features):

تعتبر إمكانية التحكم بتمثيل كيانات برنامج، من أهم ميزات لغة ADA؛ ومن دون هذا، يتوجب علينا الخروج من إطار اللغة، لإنجاز معالجة منخفضة المستوى. ولقد ذهبت ADA لأبعد من هذا، وذلك بالسماح لنا بالرجوع للميزات المرتبطة بالنظام على مستوى عال. فعلى سبيل المثال، تتضمن لغة ADA الحزمة البرمجية System، والتي تقدّم مجموعة ثوابت مرتبطة بالنظام.

وبالإضافة لذلك، يمكننا تعيين هيئات لنظامنا، من خلال استخدام عمليات (مثل System_name, Storage_Size)، أو يمكننا الرجوع لميزات ترتبط بالنظام، باستخدام واصفات (مثل Machine_Radix, Position). ولكل تنفيذ (زرع) الحق بتقديم مجموعته الخاصة من عمليات وواصفات النظام، لكن يجب عليه على الأقل، تنفيذ المجموعة الدنيا، كما سنعرف في فصل التعريفات الواردة في الملاحق A, B.

وفي بعض التطبيقات، على سبيل المثال في برامج جزئية حرجة جداً بزمن التنفيذ ، يمكن أن نكون مضطرين لكتابة ترميزنا بلغة التجميع. ويجب تجنب البدء بهكذا مستوى. فمن المفضل، تصميم النظام باستخدام بنى عالية المستوى أولاً، ومن ثمّ تسجيل، فقط، تلك الأجزاء من النظام، التي تمثل إختناقات للمنابع.

وتقدم لغة ADA أيضاً وسائل لكتابة تعليمات منخفضة المستوى. ونضع تعليمات ترميز الآلة هذه في برنامج جزئي، لا يحتوي تصريحات أو تعليمات أخرى. والحزمة البرمجية المعرّفة مسبقاً Machine_Code ، في حال وجودها، تصدر تسجيلة أو تسجيلات لتجرد مجموعة تعليمات الآلة. وبالطبع ، ترتبط هذه الحزمة بشكل أساسي بالنظام. فعلى سبيل المثال:

```
package Machine_Code is
 Bits    : constant :=1;
 Word : constant :=8;
 type Opcode is (Mov, Sub, Add);
 for Opcode'Size use 2*Bits;
 for Opcode use (Mov => 2#00#,
               Sub => 2#01#,
               Add => 2#10#);
 type Register is range 0..7;
 for Register'Size use 3*Bits;
 type Instruction is
   record
     Command    : Opcode;
     Source       : Register;
     Destination  : Register;
   End record;
 for Instruction use
   record at mod 1
     Command     at 0*Word range 0..1;
     Source        at 0*Word range 2..4;
     Destination  at 0*Word range 5..7;
   End record;
End Machine_Code;
```

```
with Machine_Code;
use Machine_Code;
procedure Copy_3 is
Begin
  Instruction'(Command => Mov, Source =>0, Destination => 1);
  Instruction'(Mov, Source => 0, Destination => 2);
  Instruction'(Mov, 0, 3);
End Copy_3;
```

بعد ذلك، يمكننا استدعاء إجرائية ترميز الآلة Copy_3، كما يلي:

```
Copy_3;
```

إن تعليمات الترميز المستخدمة في إجرائيات من هذا النوع، تأخذ شكل تعابير مقيدة. وتحتوي الحزمة البرمجية Machine_Code، فقط على تصريحات؛ والإجرائية Copy_3، لتنفذ التعليمات الموافقة، منخفضة المستوى.

١٦ - ٣ - التحويل غير المضبوط (Unchecked Conversion):

إن البنية منخفضة المستوى والصالحة بلغة ADA، تسمح لنا بتحرير قواعد تنويع اللغة. وإن استخدام هذه الوسيلة يرتبط بشدةٍ بالآلة، وبالتالي، يتعلق هذا بالمبرمج، للتأكيد من أنّ هذه الميزات تمّ استخدامها بشكل آمن وسري. وتعرّف لغة ADA إجرائيتين مولّدتين، للحصول على هذه البرمجة غير المضبوطة:

```
generic
  type Object is limited private;
  type Name is access Object;
procedure Unchecked_Deallocation ( X : in out Name);
generic
  type Source is limited private;
  type Target is limited private;
procedure Unchecked_Conversion ( S : Source ) return Target;
```

وتمثل هذه الوحدات المولّدة وحدات مكتبية، إذ يجب تسميتها بعبارة سياق من أجل استخدامها. وتقدم هذه الوحدات فائدة الصراحة، مشيرين لاستخدام طرق البرمجة غير القابلة للنقل. كما في المثال التالي:

```
with Unchecked_Deallocation;
with Unchecked_Conversion;
```

وفي حالة الإجرائية Unchecked_Deallocation ، يمكننا استخدام هذه الإجرائية من أجل التحرير صراحةً، وإعادة استخدام أماكن الأغراض المحجوزة ديناميكياً، بدلاً من السماح للنظام بحجز الذاكرة أوتوماتيكياً.

ويتمثل خطر هذه البنية، بأنّ معالجة التحرير، لا تضمن وقاية تطابق بقية أغراض الوصول التي تستطيع أن تؤشر على الغرض.

وعلى سبيل المثال:

```
with Unchecked_Deallocation;
procedure Dangerous is
  type Buffer is array(1..100) of Character;
  type Pointer is access Buffer;
  procedure Free_Buffer is new
Unchecked_Deallocation(Buffer, Pointer);
  Head : Pointer := new Buffer;
  Tail    : Pointer;
Begin
  ...
  Tail := Head;
  ...
  Free_Buffer(Head);
  ...
End Dangerous;
```

وتحرر التعليمة الأخيرة، الغرض المؤشر بواسطة Head، وتعطي القيمة Null لـ Head. وعلى أي حال، لم يتأثر Tail، وبالتالي، يؤشر على غرض غير موجود؛ وبالتالي، القيمة Tail.all غير معروفة.

وتسمح لنا الوحدة unchecked_Conversion وبحريّة، تحويل المعطيات من نوع لآخر. ويجب أن يلاحظ، بأنّ بعض المترجمات تقلص استخدامها ، معتمدةً على الأحجام النسبية لأنواع أغراض المنبع والهدف. وهذه الميزة أساسية، إذا أردنا تطبيق نوعين غير متوافقين بشكل عام فيما بينهما. وعلى سبيل المثال، غالباً ما يستخدم

Unchecked_Conversion، عند اشتراك ترميز لغة ADA، مع ترميز مكتوب بلغة أخرى. ويتمثل تأثير هذا التابع بإرجاع، سلسلة الخانات الثنائية للمعامل، وبدون مقاطعة، كقيمة من النوع Target. وعادةً لا تولّد هذه الإجرائية أي ترميز، لكنها ضرورية لتحقيق قواعد التنويع القوية في ADA. ومثال ذلك ما يلي:

```
with Unchecked_Conversion;
procedure Main is
 type Integer_16 is range -32_768..32_767;
 for Integer_16'Size use 16;
 type Word is array(1..16) of Boolean;
 pragma Pack(Word);
 for Word'Size use 16;
 function Image (Int : Integer_16) return String is
  function T_Bits is new nchecked_Conversion(Integer_16,Word);
  Bits: constant Word :=To_Bits(Int);
  Map : Constant array (Boolean)of Character := '0' , '1');
  Result : String(Bits'range);
 Begin   -- Image
  for Index in Result'range Loop
   Result(Index):=Map(Bits(Index));
  End Loop;
  return Result;
 End Image;
Begin -- Main
 ...
End Procedure Main;
```

ووفق هذه الحالة، قمنا بتحويل العدد الصحيح لمصفوفة من القيم المنطقية، من أجل سهولة توليد صورة نصية من صورة خانته الثنائية. وتُعتمد الوحدة Unchecked_Conversion، ليتم استخدامها مع أنواع المنبع والهدف من نفس الحجم. إذ أنّ نتيجة استخدام هذه الإجرائية المولّدة بأنواع من مختلف الحجوم تكون غير معرفة. وفي جميع الحالات، فهذه مسؤولية المبرمج، ليتحقق من أنّ هكذا تحويل يحافظ على خواص نوع الهدف. وعلى سبيل المثال، إن التحويلات التالية، لا تحقق ذلك:

```
with Unchecked_Conversion;
procedure Main is
 type Byte is range 0..255;
 for Byte'Size use 8;
    function  Bute_To_Character  is  new  Unchecked_Conversion(Byte,
Character);
 C : Character := Byte_To_Character(255);
 -- C Probably does not contain a Character value.
Begin -- Main
 ...
End Main;
```





مسألة التصميم الرابعة:
مراقبة البيئة
Environment Monitoring

تعريف المسألة

تحديد الأغراض

تحديد العمليات

تأسيس الرؤية

تأسيس واجهة التخاطب

تقييم الأغراض

زرع كل غرض



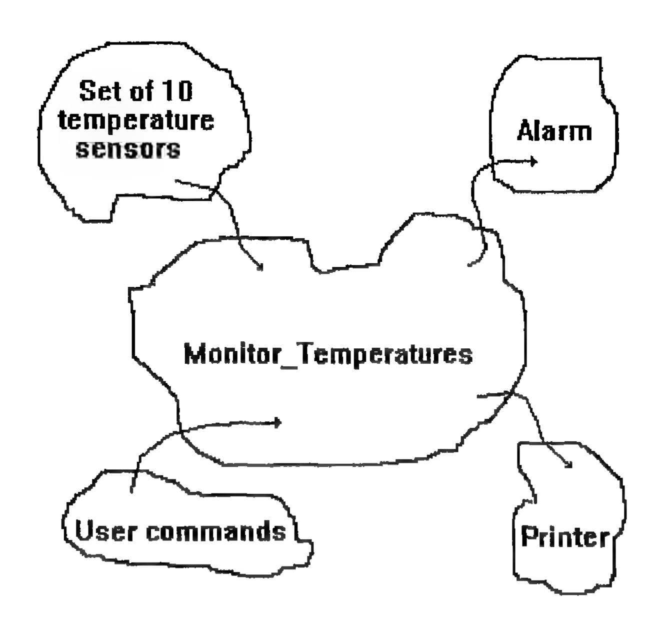
تملك معظم النظم المحمولة أربع مركبات معالجة خاصة بها هي:

- التوازي (Concurrency).
- التحكم بالزمن الحقيقي (Real-time control) .
- معالجة الاستثناء (Exception handling).
- دخل/خرج وحيد (Unique input and output).

كما شاهدنا، العديد من لغات البرمجة عالية المستوى مثل FORTRAN، Pascal وC لا تملك بُنى توافق مباشرة لهذه الحالات الخاصة. على العكس، يجب على المطور أن يستخدم التسهيلات التحتية لنظام الهدف عن طريق لغة المجمع أو استدعاءات خاصة لنظام الاستثمار. هذا يعطي برنامجا غير محمول. بشكل عام، هكذا برنامج لا يتوافق مع المسائل الضخمة. أما التطبيقات وبما أننا أكملنا دراستنا عن الأدوات المتوفرة في Ada من أجل إدارة المهام و البرمجة منخفضة المستوى، فإننا جاهزون لفحص تطبيقا في الزمن الحقيقي. في هذا الفصل، سنفحص مسألة نظام مراقبة البيئة.

١٧ ـ ١ ـ تعريف المسألة (Define the problem) :

تكون عادة نظم إدارة المعلومات مقادة من قبل الدخل/الخرج، و معظم التطبيقات العلمية مقادة بالحسابات. فيما يخص النظم المحمولة،الهم الأساسي هو التحكم أو مراقبة المعالجة بالزمن الحقيقي. و بالتالي يجب أن تستجيب النظم المحمولة غالبا إلى القيود القاسية وتكون قادرة على الإجابة وبنعومة إلى الشروط الاستثنائية كتلك التي تسببها أعطال التجهيزات.سنفحص في هذا الفصل مسألة نظام مراقبة البيئة. وكما في باقي المسائل طريقة غرضية التوجه لإيجاد حل كامل بلغة ADA. الشكل١٧ ـ ١ يوضح فضاء المسألة.بالرغم من أن هذا النظام يستخدم مراقبة مجسات الحرارة فمن الواضح أن هذا الحل يمكن أن يمتد إلى باقي الأنظمة التي تستخدم مجسات أخرى كالتي تسمح بمراقبة الضغوط والفولتاج (الجهد) أو مستوى السوائل.



الشكل ١٧ ـ ١. مسألة مراقب البيئة.

تتطلب مسألتنا عدة مجسات حرارة في مواضع مختلفة من بناء ما. فالمجسات تأخذ عينات وبشكل مستمر لدرجة حرارة المحيط فإذا اكتشف مجس ما درجة حرارة أعلى من حد معطى فإن النظام يطلق إنذارا. المجس يظهر أيضا درجة الحرارة إذا كانت ضمن المجال. وبالتالي المستخدم يمكن أن يتفاعل مع النظام بإعطاء حدود الإنذار لكل مجس وبقراءة حالة جميع المجسات. عندما يطلق الإنذار يمكن للمستخدم وبسرعة تحديد موضع شرط درجة الحرارة غير النظامية. وبشكل دوري يجب على نظامنا أن يطبع القيم الجارية لجميع المجسات على لائحة وبشكل مستمر. للطابعات عادة سيئة هي أنها تحتاج دائما إلى ورق وخاصة في الأوقات الحرجة ولذلك نعتقد بأن الطابعة هي المحيطي الأقل وثوقية في نظامنا. وعلى نظامنا أيضا أن يكون قادرا على كشف خطأ المحيطي ويطلق صفارة الإنذار.

لنفترض الآن أن تطبيقنا يدور حول حاسب هدف يستخدم المداخل– المخارج في أماكن الذاكرة. يمكننا إذا أخذ عينات بقراءة المجسات على عنوان ذاكرة خاص أو يمكن إطلاق الإنذار بالكتابة إلى موضع معين. لم نعف بعد أي قيد أداء. سنفحص هذه النقطة خلال تحليلنا .

١٧ ـ ٢ ـ تحديد الأغراض (Identify the Objects) :

بفحص الأسماء التي استخدمناها في وصف فضاء المسألة، يمكن استخلاص مباشرة الأغراض وصفوف الأغراض التي تهمنا لتحليلنا. لقد ذكرنا بشكل خاص :

- مجموعة مجسات .
- طابعة .
- إنذار من أجل شروط التجاوز وأخطاء الطابعة .
- منافذ الإدخال والإخراج في الذاكرة .
- مستخدم .

تعرف مجموعة المجسات في الواقع صف أغراض. بينما لا يتطلب نظامنا إلا طابعة وإنذار كآلات مجردة . يمكن التعميم أكثر من ذلك. بشكل خاص تشترك جميع المجسات بشيء واحد: جميعها تقيس واصفات فيزيائية من عالم حقيقي. فمن الممكن إذ كما سنرى، تجريد المجس مباشرة من نوع القياس الفيزيائي الذي يذود به. وتجريدنا لمجس يطبق على مجسات الحرارة ومجسات الضغط ومجسات مستوى السوائل على حد سواء. لا يمكننا فقط تصنيف هذه المجموعة من المجسات كنوع مجرد من المعطيات لكن معالجة حلنا كتجريد مولد أيضا من أجل تطبيقات خاصة إذ يمكننا إعادة استخدام هذا التجريد من أجل تطبيقات أخرى إذا احتجنا إليها. في الواقع بقدر ما يكون هناك تغيرات في دفتر الشروط للمسألة الحالية فإن التصميم يجعل تعديل حلنا أكثر سهولة. لاحظ كذلك أننا جمعنا منافذ الإدخال والإخراج في الذاكرة بغرض تجريد وحيد. هذا ليس تجميعا منطقيا فقط لكن كما سنرى تعزل هذه الطريقة معظم مواصفات حلنا التي تتعلق بالزرع. العزل مرغوب لأننا إذا أردنا حمل تطبيقنا على آلة هدف مختلفة فيجب تغيير التطبيق.

يجب أخيراً وضع المستخدم جانباً والذي هو غرض مختلف قليلا عن جميع الأغراض الأخرى لفضاء المسألة. فالمستخدم موجود خارج تطبيقنا، والمستخدم يمكنه التفاعل مع النظام عبر أوامر. لذلك لن يحوي حلنا إذاً غرضا برمجيا مشابها لهذا التجريد العضوي. ولذلك فإن بنية التحكم الأساسية تنوب عن واجهة التخاطب

المستخدم وبذلك يتم التعامل مع الأنظمة المقادة بواسطة المستخدم. وستبقى بقية الأغراض على شكل كيانات متمايزة في بنية حلنا.

١٧ ـ ٣ ـ تحديد العمليات (Identify the Operations) :

في هذه المرحلة سننظر إلى سلوك جميع الأغراض من وجهة نظر خرجين ولن نحدد العمليات التي تخضع لها الأغراض لكن يجب تحديد التوازي لكل منها أيضا. وبشكل خاص يجب أن نقرر فيما إذا كان كل غرض هو عامل Actor أو محول Transducer أو مخدم Server وسنرى لاحقا أن هذا القرار هام لأنه يؤثر على ترابط الأغراض.

سوف نجرد مجس وكأنه كيان مواز ودوره الأساسي هو مراقبة درجة حرارة مكان وبشكل مستمر. وبما أن لدينا العديد من المجسات وبما أن كل مجس يقيس وبشكل مستمر فهذا يتطلب ضرورة بنائها على شكل مهام. خيار آخر يتمثل بمعالجة كل مجس وكأنه كيان سلبي ولا يوجد سوى عملية معالجة (Process) وحيدة لاختبارها. هذه هي الطريقة التقليدية المستخدمة باللغات القديمة مثل الفورتران والباسكال. في حين أن استخدام مهام ADA تسمح لنا بضبط حقيقة طبيعة التوازي لفضاء المسألة وإدخالها بشكل طبيعي في حلنا. تعطينا ADA أيضا مرونة كبيرة للتحكم بالسلوك الزمني لنظامنا .لنعتبر الآن العمليات التي يمكن أن يخضع لها مجس. للوهلة الأولى يمكن الظن بأن مجسا ما لا يخضع لأية عملية بل بالأحرى مهمة عاجلة. ويمكن أن يكون هذا حلا معقولا. ومع ذلك سنترك للزبون مسألة مراقبة المجسات وسوف توصف المجسات وكأنها محولات. فمن جهة يستخدم المحول عمليات بقية الأغراض يمكن مثلا إطلاق إنذار ومن جهة أخرى وبما أننا نسمح بتداخلات المستخدم فيجب أن نكون قادرين على تنظيم حدود الإنذار لمجس ما ومراقبة حالة المجس أي يجب أن نكون قادرين على الحصول على قيمة درجة الحرارة الجارية وإذا كانت موجودة ضمن الحدود أم لا. يجب إذاً إدراج هذه النشاطات كعمليات أغراض مجسات. وبذلك نسمح بالعمليات التالية :

- **Start** بدء نشاط المجس.
- **Set_Limit** إعطاء قيمة لإقلاع الإنذار.
- **Get_Status** إرجاع القيمة الجارية للمجس.
- **Shut_Down** إيقاف عمل المجس.

إن وجود **Set_Limit** و **Get_Status** ليس مدهشاً لكن لماذا يجب علينا إدراج عمليات الإقلاع وإيقاف المجس بشكل صريح. كما قدمنا في الفصل ١٤ فإن ADA تدخل قواعد خاصة لتنشيط وإنهاء المهام. مع ذلك نريد بشكل صريح، في مثالنا مراقبة مدة حياة جميع الأغراض. ولهذا السبب قدمنا **Start** و **Shut_Down**.

إن ضرورة وجود **Shut_Down** يجب أن تكون واضحة : فكما عرضنا في الفصل السابق فإن **Shut_Down** تمثل الطريقة التقليدية للسماح بإنهاء المهمة؛ نحن بحاجة أيضاً لـ **Start** لأنها في الواقع لا تنشط المهمة. لأن قواعد ADA تقضي بأن يكون التنشيط صريحاً، لكن نستخدم **Start** لمنع الأغراض من القيام بأي عمل كان قبل أن نكون جاهزين للإقلاع. أكثر من ذلك فإن هذه العملية تحل مسألة إطلاع مهمة ما على اسمها الخاص بها.بما أننا اخترنا توصيف المجسات كأغراض مستمدة من صف المجسات فإن حلنا يجب أن يكون قادرا على التمييز بين عدة أغراض مهام. فكل مجس يجب أن يقيس درجة الحرارة في مكان مختلف.

لقد اخترنا أيضا توصيف صف المجسات هذا كمولد وعلينا كذلك الأخذ بعين الاعتبار العمليات المطلوبة من مجس ما. تساعدنا هذه الطريقة بتحديد ماذا يجب أن تستورد وحدتنا المولدة. تتطلب استراتيجيتنا معرفة المتطلبات المشتركة التي يمكن تحديدها بين جميع الأغراض المجسات، ومن ثم تجريدها كأنها عوامل مولدة. كما قلنا في فصل سابق نستورد بشكل خاص :

- **Name** نوع مستخدم لتحديد مجس خاص.
- **Value** نوع يحدد صف القيم المقاسة .
- **Sense_Rate** فترة أخذ المجس لعينات البيئة .

أكثر من ذلك يجب أن نستورد عملية تسمح لمجس ما بالذهاب لقراءة منفذ إدخال / إخراج في الذاكرة وعملية كهذه تطلق الإنذار. بتدوير غرضنا هكذا نكون فعلاً قد فصلنا تجريد المجس عن بقية جميع أغراض فضاء المسألة.

لنعتبر الآن سلوك الطابعة. فهي غرض وحيد موصف وكأنه آلة مجردة. سنعامل الطابعة ككيان مواز ما دامت تتفاعل مع بقية المهام. إن الحاجة لتحديد كيانات موازية هي نتيجة مألوفة للأنظمة الموازية في ADA، فمن الصعب كتابة مهمة واحدة فقط. اعتباراً من اللحظة التي يقرر فيها المطور إدخال التوازي، يجب اعتبار سلوك جميع الأغراض بحضور مهام مضاعفة. فمثلاً ليس من المؤكد تعليب محيطي كطابعة على شكل كيان تسلسي بسبب مشاكل الاستبعاد المتبادل.

الطابعة هي غرض مخدم بشكل كامل فلا تستدعي أي عملية.ومن الخارج نسمح للطابعة أن تخضع إلى العمليات التالية:

- **Put_Line** طباعة سطر
- **Shut_Down** إيقاف عمل الطابعة

كما عملنا من أجل المجسات فقد أدرجنا عملية تسمح بمراقبة توقف هذه المهمة بشكل صريح ولسنا بحاجة لإدراج عملية Start ، فلا يوجد سوى غرض واحد، وبالتالي لسنا مجبرين لتمرير اسمه إلى هذه المهمة.

الإنذار له مواصفات مماثلة لمواصفات الطابعة. يمكننا تجريد الإنذار كغرض مخدم مزود بالعمليات التالية :

- **Report_Out_Of_Limits** إطلاق الإنذار من أجل درجة حرارة غير طبيعية.
- **Report_Priner_** إطلاق إنذار الطابعة.
- **Shut_Down** إيقاف الإنذار.

مرة أخرى لا يوجد سوى غرض الإنذار. يلزمنا إذاً عملية **Shut_Down** وليس عملية **Start**.

لاحظ أننا لم ندرج عملية من أجل إيقاف صفارة الإنذار. ونفترض للحين أن هذا خارج نطاق حلنا البرمجي. افترض أن تطبيقنا لا يحتاج إلا لوصف قيمة لكل موضع خاص في الذاكرة من أجل إطلاق صفارة الإنذار وإن على المستخدم إيقافها.

لنعتبر الآن آخر غرض رئيسي لفضاء المسألة منافذ الإدخال/الإخراج. من الواضح أن هذا الغرض يقع في مستوى تجريد أضعف من جميع مستويات الأغراض التي فحصناها، ونتكلم عنه مع ذلك لأنه عنصر هام من فضاء المسألة. يطلب هذا الغرض عدة أغراض أخرى – مثل منافذ الدخل/الخرج لجميع المجسات و الإنذار. هذا النوع من التعليب هو في الواقع مألوف جداً في الأنظمة المحمولة. في منظور لغة التجميع من المفيد تحديد الـ Macros التي تقنع استخدام العناوين الفيزيائية. وتجريدنا هو المكافئ للمستوى العالي للغة التجميع. وهذا الغرض لا يستورد إذاً أي عملية بشكل صريح، لكن ضروري فقط لإعطاء قيم للمتغيرات بسيطة وإيجادها من جديد.

١٧ ـ ٤ ـ تأسيس الرؤية (Establish the Visibility) :

الآن وقد وصفنا سلوك جميع الأغراض الأساسية لفضاء المسألة يجب أن نأخذ بعين الاعتبار علاقاتهم. بشكل خاص يمكن أن نؤكد أن تجريدنا للمجس معزول بشكل كامل عن باقي الأغراض. مع ذلك فإن نسخ هذا المركب مرتبط بتجريديات أخرى. فإذا سمينا النسخ Temperature_Sensors :

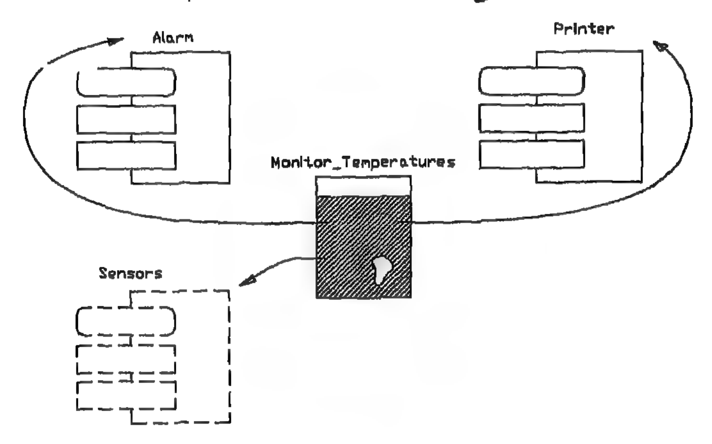
Temperature_Sensors يجب أن يرى **Printer and Alarm**

وبالطبع العكس ليس صحيحاً. فلا Alarm ولا Printer يمكنهما رؤية النسخ وكذلك انهما منفصلان أحدهما عن الآخر.

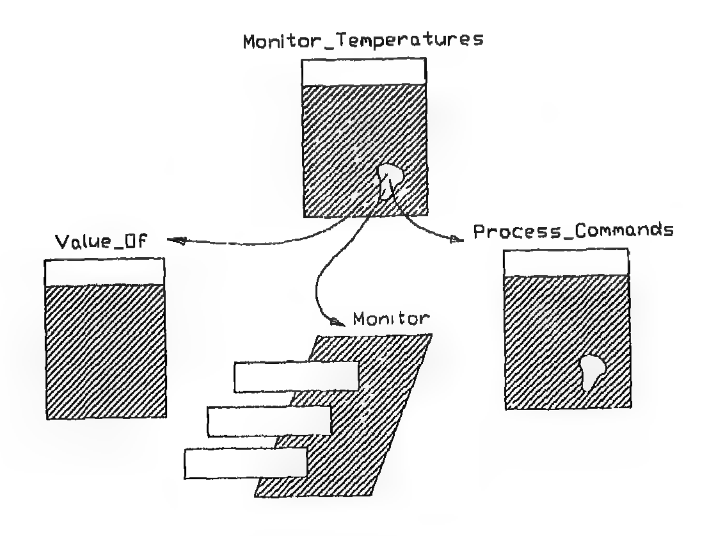
يجب أن يحتوي تطبيقنا، كما تتطلب ADA برنامجاً جزئياً أساسياً يستخدم كجذر للنظام. نسمي هذا البرنامج الجزئي Monitor_Temperatures ويحتوي على نسخ المجسات. أكثر من ذلك ستكلف هذه الوحدة بتفاعلات المستخدم. وبالتالي :

Monitor_Temperatures يجب أن ترى **Printer and Alarm**

يوضح الشكل ١٧ ــ ٢ هذه العلاقات. لقد استخدمنا رمزاً خاصاً في جسم البرنامج الرئيسي لنشير إلى أن هذه الوحدة تتفكك من جديد إلى وحدات جزئية .وإذا نظرنا بعدسة مكبرة على جسم Momitor_Temperatures نرى أنه يحتوي ليس فقط على نسخ المجسات لكن أيضاً على ثلاث وحدات جزئية هي : برنامج جزئي ليسمح بالنسخ، برنامج جزئي ليعالج تفاعلات المستخدم ومهمة لتعمل التسجيل الدوري لقيم المجسات. الشكل ١٧ ـ ٣ يوضح هذا المستوى من التصميم.

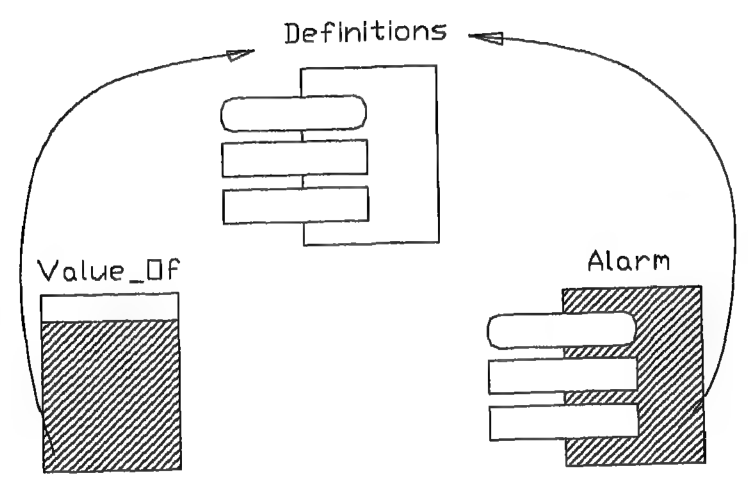


الشكل ١٧ ـ ٢. تصميم Monitor_Tempretures.



الشكل ١٧ ـ ٣. جسم الـ Monitor_Tempretures.

القارئ الحريص سيسأل ماذا حدث لمنافذ الدخل/الخرج في الذاكرة .كما شرحنا يوجد هذا التجريد في أخفض مستوى من نظامنا. كما سنرى فالشكل 4.17 يوضح أن هذه المنافذ (التي وضعناها في حزمة برمجية تدعى Definitions) ليست مرئية إلا لأجسام الوحدتين .



الشكل ١٧ ـ ٤.

١٧ ـ ٥ ـ تأسيس واجهة التخاطب (Establish The Interface):

نحن الآن مستعدون لأخذ قراراتنا باستخدام ADA كلغة تصميم. لنبدأ بعرض بسيط ونتقدم فيما بعد. يمكن التعبير عن الرؤية الخارجية للطابعة بتوصيف الحزمة البرمجية :

```
Package  Printer is
  task The_Printert is
    entry Put_Line ( The_Item : in String ) ;
    entry Shut_Down;
  end The_Printer;
end Printer;
```

لاحظ بأي طريقة عملنا للتصريح مباشرة عن غرض مهمة. كما ذكرنا في الفصل١٤ ، فإن قواعد ADA تقضي بأن تكون المهام مصرح عنها في سياق وحدة شاملة

مثل الحزمة البرمجية المكتبية التي لدينا الآن. وليس من الضروري تصدير أي شيء آخر. مداخل المهمة تقدم العمليات المطلوبة بتجريدنا.

طريقة أخرى تقضي تخبئة المهمة في جسم الحزمة البرمجية وعدم تصدير إلا البرامج الجزئية التي تستدعي بدورها المداخل الموافقة. وهذا بالضبط اخترناه في مثالنا للحزمة البرمجية Spooled_Print في الفصل ١٤. لن نستخدم طريقة العمل هذه هنا للسماح لزبائن الحزمة البرمجية باستخدام استدعاءات مدخل Limited_Delay. الرؤية الخارجية للإنذار مشابهة إلى رؤية الطابعة:

```
Package Alarm is
 task The_Alarm is
entry Report_Out_Of_Limits;
entry Report_Printer_Error;
entry Shut_Down;
 end The_Alarm;
end Alarm;
```

واجهة تخاطب المجسات مشابهة ما عدا الوحدة المولدة المرجعة والعناصر الخاصة المستوردة مسبقاً: إن التصريح من نوع مهمة **Sensor** تشبه ما صرحنا به من أجل **Printer and Alarm** بالطبع لدينا هنا نوع مهمة وليس غرض مهمة. علينا إذاً التصريح عن أغراض نوع. سنصرح عن هذه الأغراض في البرنامج الرئيسي. الوحدة لا تصدر أغراض مهمة مما يسمح لنا بتأخير القرار بالنسبة لعدد المجسات الموجودة فعلاً. وإذا تغير دفتر الشروط للحصول على عدد مختلف من المجسات فيجب ألا نغير الوحدة المولدة هذه.

وكون القسم المولد لهذه الحزمة البرمجية قليل التعقيد فسوف نفحصه بالتفصيل. وكما في مولد الأشجار في الفصل ١٣، فقد استوردنا الأنواع **Name and Value**. لكن قيدنا هنا هذه الأنواع لتكون على الترتيب منفصلة وصحيحة. وهذا يعني (تذكر ما كان في الفصل ١٢) أننا نستورد ضمنياً مجموعة عمليات منفصلة وصحيحة (كما في الإسناد واختبار المساواة) وكذلك معاملات علائقية (وغير ذلك بالطبع). العامل المولد **Sense_Rate** هو عامل قيمة مولد. يجب أن نوافق له قيمة من النوع

Duration مشيراً بذلك إلى التواتر الذي يجب أن يقرأ به المجس درجة الحرارة المحيطة. أخيراً إن **Value_Of and Sound_Alarm** هي عوامل برامج جزئية مولدة. إن استخدام هذه العوامل يسمح بفصل الوحدة فعلياً عن كل آلية الإنذار أو أي قياس خاص. وبهذا يكون من الممكن إعادة استخدام هذا التجريد من أجل تطبيقات أخرى.

وبما أن هذا التجريد يوجد في أخفض مستويات التجريد فسوف نأخذ بعين الاعتبار الرؤية الخارجية لمنافذ الدخل /الخرج في الذاكرة. وكما أشرنا فإن هذا التجريد هو في الواقع رؤية من المستوى العالي لموضع الذاكرة الفيزيائية. واضح أن هذا الموضع لمنافذ الذاكرة يتعلق بالآلة الهدف. وبتوصيف هذه القيم لوحدها في هذه الوحدة البرمجية نعزل بالفعل كل ارتباط بالهدف عن باقي النظام. لحسن الحظ توصيفات التمثيل بلغة ADA قوية بشكل كاف لتسمح بتوصيف الحجم والعنوان المضبوطين لكل من هذه النوافذ. من أجل احتياجات هذه المسألة، سنفترض أن موضع هذه النوافذ قد قدم من قبل مصممي التجهيزات. وبالتالي يمكن التعبير عن الرؤية الخارجية لهذا التجريد كما يلي :

```
package Definitions is
type Byte is range 0..255;
for Byte 'Size use 8;
package Sesor_Ports is
Lobby , Main_Office, Warhouse, Stock_Room , Terminal_Room,
Library, Computer_Room, Lounge, Loadding_Dock, Clean_Room:
Byte;
for Lobby          use at 16#30#;
for Main_Office    use at 16#32#;
for Warehouse      use at 16#34#;
for Stock_Room     use at 16#36#;
for Terminal_room  use at 16#38#;
for Library        use at 16#3A#;
for Computer_Room  use at 16#3B#;
for Lounge         use at 16#3C#;
for Loading_dock   use at 16#3D#;
for Clean_Room     use at 16#3E#;
```

```
end Sensor_Ports;

    package Alarm_Ports is
On : constant Byte := Byte'Last
Out_Of_Limits,Printer_Error : Byte;
for Out_Of_Limits use at 16#80#;
for Printer_error   use at   16#100#;
end Alarm_Ports;
end Definitions;
```

لاحظ استخدام تصريحات الحزم البرمجية المتراكبة لتجمع منطقياً صفي النوافذ الأساسيين. مع عبارات التمثيل التي قدمناها يمكن للزبون إسناد قيمة إلى الغرض Printer_Error لوصف فعلياً ثمانية بتات للعنوان الفيزيائي #16#100. لاحظ أيضاً استخدام الأعداد المعتمدة للتعبير عن مواضع الذاكرة بشكل أكثر عملي وذلك من أجل الرؤية المادية للتجريد. كذلك بالإرجاع إلى الغرض Lobby فإن تطبيقنا يقرأ فعلياً ثمانية بتات بالعناوين #16#30 .

١٧ - ٦ - زرع كل غرض (Implement Each Object) :

بعد معالجة الرؤية الخارجية لجميع التجريدات ذات المعنى، سوف نتوجه إلى الرؤية الداخلية. وبما أننا درسنا الحزمة البرمجية Definitions فسوف نفحص جسم الوحدة البرمجية المعتمدة على هذا التجريد. يتطلب الجسم Alarm بشكل خاص أن نكتب في المنافذ Alarm_ports المعرفة سابقاً. مع ذلك فإن هذا الجسم هو أعقد بقليل لأنه يجب علينا الأخذ بعين الاعتبار بوجود مهمة.

صحيح أنه في العديد من التطبيقات في الزمن الحقيقي يكون وجود إجرائيات تدور بشكل لا نهائي مألوفاً. لذلك يمكن كتابة جسم المهمة Alarm كحلقة بسيطة تتضمن تعليمة Select متراكبة. في هذه الحالة يختار الزبون مدخلاً من بين ثلاثة كما توضحه Select المتراكبة .

```
with Definitions;
package body Alarm is task body The_Alarm is
On : Definitions.Byte renames Definitions.Alarm_Ports.On;
```

```
 begin
   loop
      select
        accept Report_Out_Of_Limits := On;
      or
        accept Report_Printer_Error;
      or
        accept Shut_Down;
        exit;
      end select;
    end loop;
  end The_Alarm;
end Alarm;
```

الحزمة البرمجية Alarm التي تستخدم الموارد Definitions يجب ذكرها في عبارة السياق. لاحظ كيف عالجنا تزامن المهام. إن الكتابة في المنفذ مدخل/مخرج تتم خارج التعليمة accept بشكل نحرر المهمة الداعية مباشرة بعد قبول الموعد. هذا عمل شائع للمداخل بدون عوامل. لاحظ كذلك أنه لإطلاق الإنذار لسنا بحاجة إلا لإسناد قيمة واحدة لنافذة معينة مما يؤدي إلى كتابة في هذا الموضع من الذاكرة.

الحزمة البرمجية Alarm تعالج التوقف باستدعاء صريح للمدخل Shut_Down. عندما يقبل هذا المدخل فإن Alarm تخرج من حلقتها مؤدية إلى إكمال سلسلة التعليمات المهمة وبالتالي التوقف.

جسم الحزمة البرمجية Printer يشابه كثيرا جسم Alarm ما عدا أننا حذفنا الكود الذي يرسل فعلياً السلسلة إلى المحيط الفيزيائي. المدخل Put_Line يملك عاملاً. التعليمة accept الموافقة تتضمن إذاً سلسلة من التعليمات لمعالجة قيمتها :

```
with Text_IO;
package body Printer is
   task body The_Printer is
     type String_Access is Access;
     Local        : String_Access;
     Print_File  : Text_IO.File_Type;
begin
```

```
  Text_IO.Create(File => Print_File, Name => "PRN" : );
                  --         system dependent
loop
      select
        accept Put_Line ( The_Item  : in String  )  do
          Local  := new String'( The_Item  );
        end Put_Line;
        Text_IO.Put_Line( Print_file, Local.all );
    or
        accept Shut_Down;
        exit;
      end select;
    end loop;
    Text_IO.Close  ( Print_File ) ;
  end  The_Printer;
end Printer;
```

مع ذلك يجب أن نكــون حذريـن عنـد كتابـة هـذا الرمـز. إذا كـان تنفيـذ هـذه التعليمات مؤجلاً بشكل غير محدد فإن المهمة الداعية ستكون مؤجلة أيضاً وهذا موقف غير مرغوب فيه.ولتفادي ذلك هناك وسيلة تقضي الحفاظ علـى قيمـة العـامل والخـروج من الموعد مباشرة ثم معالجة السلسلة.وهكذا لا يمكن تأجيل المهمة المسـتدعية بشـكل غير محدد.

الآن وقد استعرضنا أجسام المهام هذه، فإن جسم الحزمة البرمجية Sensor لا يبدو مقلقاً. إن العمل الأساسي لمهمـة المجـس هـو قيـاس درجـة الحـرارة المحيطيـة باستمرار وذلك عن طريق أخذ عينات كل Sense_Rate ثانية. إن مركز هذا الجسم هو إذاً حلقة تحتوي على التعليمة Select المتراكبة وفق بنية متشابهة لبنية Printer and Alarm والفرق هو أننا نريد أن يحصل شيء بشـكل دوري وليـس فقـط عنـد اسـتدعاء مدخل.

الحل الذي سنستخدمه يتضمن حلقة مؤخرة تشبه ما اسـتخدم فـي الفصـل ١٤. تذكر أنه يمكن استدعاء Set_Limit,Get_Status or Shut_Down فـي الوقـت ذاتـه. لأنه لا نريد جعل الزبائن ينتظرون. لتلافي التأخير نستخدم تعليمة Select مـع خيـار

التأخير والتي نحسب فيها التأخير عند كل مرور في الحلقة. وهكذا في بداية الحلقة نحسب الزمن التالي الذي نقيس فيه المحيط. خوارزميتنا تنتظر في بداية التعليمة Select. فإذا استدعي مدخل فإنها تقبله ولكن إذا لم يحدث أي شيء في الزمن الذي حسبناه فإن التأخير يتم انتقاؤه والمجس يقيس. نحدد في هذا الوقت كذلك فيما إذا كان المجس خارج الحدود وأننا نطلق الإنذار في حال الحاجة.

من الواضح أن كل مجس يجب أن يحفظ حالة ما , Current_value) The_Limit, or Next_Time) معلنة متى يجب القياس. أكثر من ذلك فإن مهمة مجس يجب أن تعرف اسمها. إن تسمية مهمة ما هي مسألة تصميم نموذجية ل ADA. فيجب ألا نكتب جسم مهمة واحد فقط لكن كل نسخة عن مهمة تفعل شيئاً مخالفاً قليلاً. في تطبيقنا الاختلاف يكمن في فهم أن كل مهمة تأخذ عينات منفذ فيزيائي مختلفة. في توصيف الحزمة البرمجية Sensor استوردنا العملية Value_Of، لكن الداعي لهذه العملية يجب أن يكون قادراً على تعريف نفسه. كذلك عندما نقلع Start المهمة التي يجب أن نعطيها اسماً (قد يكون اسم المكان الذي تشغله). ثم نمرر فيما بعد هذا الاسم إلى التابع Value_Of، عند استدعائه حتى يمكنه البحث عن المنفذ الجيد. لذلك يجب على جميع المهام أن تحتفظ بـ Sensor_Name، التي هي حالة معلومات أخرى. بهذه القيود على سلوك جسم المهمة يمكننا كتابة :

```
with Calander;
 package body Sensors is
  task body sensor is
    Sensor_Name        : Name;
    Current_Value      : Value := Value'First;
    The_Limit          : Value := Value'Last;
    Next_Time     : Calendar.Time := Calender.Clock;
    use Calender;
   begin
    select
     accept Start ( The_Name : in Name ) do
       Sensor_Name := The_Name ;
```

```
      end Start ;
     or
      terminate;
     end select;
         loop
      select
       accept Set_Limit ( The_Value :  in  Value  )  do
        The_Limit  :=  The_Value;
       end Set_Limit;
      or
       accept Get_Status  (  The_Value     : out Value  ;
                  Out_Of_Limits:out Boolean  )    do
        The_Value    :=   Current_Value    ;
        Out_Of_Limits := Current_Value   >   The_Limit ;
       end Get_Status;
      or
       accept Shut_Down;
       exit;
      or
       delay  Next_Time - Calendar.Clock;
       Current_Value   :=  Value_Of ( sensor_Name  );
      If Current_Value>The_Limit then  Sound_Alarm
       end if ;
Next_Time := Next_Time+Sense_Rate;
End select;
End loop;
End Sensor;
End Sensors;
```

تشبه الحلقة في نهاية مهمة Sensor كثيراً حلقة The_Printer and The_Alarm لكن تملك Sensor خيار التأخير delay. لاحظ مع ذلك أن تعليمة ال accept من أجل مهمة Start هو جزء من تعليمة Select المختلفة. لماذا ؟ لأن هذا الدخل لا يستدعى إلا مرة واحدة ولا نهتم به طيلة فترة حياة الغرض. وبما أننا نريد

حماية أنفسنا من حوادث خطيرة فقد وضعنا الخيار treminate. فإذا لم تقلع مهمة نسخ ما يمكنها التوقف بنعومة.

نحن الآن جاهزون لبناء جسم البرنامج الرئيسي (Monitor_Temperatures). تذكر أن هذه الوحدة تقدم نسخاً من الحزمة البرمجية Sensors وكذلك كل التفاعلات مع المستخدم.

```
With Sensors, Printer , Alarm;
Procedure Monitor_Temperature is
  Type Sensors_Names is( Lobby, Main_Office,Warehouse,
  Stock_Room, Terminal_Room, Library ,
         Computer_Room , Lounge, Loading_Dock,Clean_Room);
  Subtype Value is Natural;
  Once_per_Second : constant Duration := 1.0 ;
Function Valur_of( The_Name : in Sensor_Names ) return
    Value is separate;
Package Temperature_Sensors is new Sensors
    ( Name      => Sensor_Names,
      Value     => Value,
      Sense_Rate => Once_per_second,
      Value_Of  => Value_Of,
     Sound_Alarm => Alarm.Thw_Alarm.Report_out_Of_Limits);
      The_sensors : array (Sensor_Names) of
         Temerture_Sensors.Sensor;
Function Format ( Name      : Sensor_Names;
      Reading   : Value ;
            Out_Of_Limits : Boolean )

        Return String is separatee ;
  task Monitor is
      Entry Shut_Down;
End Monitor;
                Task body Monitor is separate;
Procedure process_Commands is separate;
 Begin -- Monior_Tempertures
                For Index In Sensor_Names
```

```
                Loop
                      The_Srnsors(Index).Start(The_Name  => );
                End loop;
        Process_Commands;
               Monitor.Shut_Dwon;
               For Index in Sensor_Names
                      Loop
                        The_Sensors(Index).Shut_Dwon;
                      End loop;
               Printer.The Printer.Shut_Dwon;
               Alarm.The_Alarm.shut_Dwon;
End Monitor_Tempertures;
```

الجزء الضخم من هذا البرنامج الرئيسي مؤلف من قسمه التصريحي. نقدم أولاً عدة تصريحات (Sensor_Names , Value, Once_Per_Second , and Value_Of) ضرورية لنسخ الحزمة البرمجية Sensors. نصرح فيما بعد عن مصفوفة مجسات (The_Sensors). يتبع ذلك مهمة Monitor التي تسجل وبشكل دوري قيمة جميع المجسات، كما يتبع أيضاً إجرائية تعالج التفاعلات مع المستخدم.

تسمح التعليمات المرتبطة بهذا البرنامج تنشيط وإنهاء المهام بالترتيب الصحيح. نعلم أن المهام Printer and Alarm المستوردة بالعبارة with ستنشط قبل إعداد القسم التصريح للبرنامج الرئيسي. وهذا أكيد لأنها ليست سوى مهام خدمة. المهام The_Sensors نشطت كلها في نهاية إعداد القسم التصريحي للبرنامج الرئيسي. وهذا ينطبق أيضاً على مهمة Monitor. بالرغم من أن مهام The_Sensors تكون منشطة عند هذه النقطة فهي كلها مؤجلة على accept لـ Start. كذلك في جسم Monitor_Temperatures باستدعاء مدخله Start نستدعي Process_Commands. بعد العودة من هذه الإجرائية (عندما يوقف المستخدم النظام) يجب أن نوقف المهمة Monitor لمنع متابعة طباعة العينات. ثم نوقف جميع المجسات ثم المهام Printer and Alarm. إذا أوقفنا Printer and Alarm في وقت أبكر فهذا يسبب مشاكل.

بشكل خاص إذا حاول الـ Monitor طباعة قياسات في حين أن المهمة Printer تكون انتهت. فإن الـ Monitor سيستقبل الاستثناء Tasking _Error وينهـي العمـل بشـكل عنيف.

```
with Calendar;
separate ( Monitor_Temperatures )
Function Format ( Name          : Sensor_Names
            Reading              :  Value ;
            Out_Of_Limits: Boolean ) return String is
 Message  : array ( Boolean ) of String ( 1.. 15 )  :=
                  ( False => ( other  =>  Ascii.Nul  ) ,
                  ( True  => ": OUT  OF LIMITS"  );
 ' funcion Pad_Left ( Item :String; Length : Positive)return String is
    begin
     return  String'(1..Length  -  Item'Length  =>  ' ' ) ;
     end Pad_Left;
     funcion Pad_Right ( Item : String ;Length  : positive)return String is
 begin
  return Item & String ' (1.. Length - Item'Length => ' ');
 end Pad_Right;
function Format(The_Time:Calender.Time)  return String is separate;
 begin
return Format  ( Calendar.Clock)  & ' '                    &
Pad_right(Sensor_Names'Image(Name),Sensor_Names'Width &
"  sensor value is "                                        &
Pad_left ( Value'Image ( Reading) , Value'width          &
"   degrees" & Message ( Out_Of_Limits) ;
end Format;
```

لقد استخدمنا وحدات جزئية في هذه الإجرائية لتسهيل التنفيذ. سوف نفحـص الآن هذه الوحدات الجزئية. Value_Of تعمل كجسم Alarm وإن تعليمة Case تبحث عن منفذ مدخل/مخرج الموافقة لاسم المجس :

```
with Definitions;
separate  ( Monitor_Temperatures )
funcion Value_Of  The_Name : in Sensor_Names ) return Value is
```

```
begin
case The_Name is
when Lobby =>
return Value(Definitions.Sensor_Ports.Lobby);
when  Main_Office  =>
return   Value(Definitions.Sensor_Ports.Main_Office);
when   Warehouse        =>
return
Value Definitions.Sensor_Ports.Warehouse);
when Stock_Room          =>
return Value(Definitions.Sensor_Ports.Stock_Room);
when Teminal_room         =>
return
Value( Definitions.Sensor_Ports.Terminal_Room);
when Library
return
Value( Definitions.Sensor_Ports.Library);
when Computer_Room        =>
return
 Value( Definitions.Sensor_Ports.Computer_Room);
when Lounge              =>
return
 Value  ( Definitions.Sensor_Ports.Lounge);
when Loadding_Dock        =>
return
 Value( Definitions.Sensor_Ports.Loading_Dock);
when Clean_Room           =>
return
 Value ( Definitions.Sensor_Ports.Clean_Room );
end case;
end Value_Of
```

لاحظ أنه يجب استخدام التحويل ما بين الأنواع في كل تعليمة return لأن المنافذ هي من نوع Byte ومن النوع Value.

جسم المهمة Monitor يستخدم حلقة مؤخرة لتأخذ العينات وبشكل دوري من كل المجسات وتطبع حالتها. يجب أن نحمي أنفسنا من مشاكل الطابعة. لقد قلنا سابقاً أن المهمة Printer يمكن أن تؤجل بشكل غير محدد. بدل أن نجعل المهمة Monitor تنتظر بشكل غير محدد سنستخدم استدعاء مدخل بانتظار محدد بشكل أن الـ Monitor يمكنه المتابعة إذا انتظر طويلاً. يمكننا الكتابة إذاً :

```
with Calender;
separate ( Monitor_Temperatures )
 task body Monitor is
  Seconds      : constant Duration := 1.0;
  Time_Interval: constant Duration := 60*Seconds;
  Next_time    : Calender.Time := Calender.Clock;
  The_Value    : Natural ;
  Out_Of_Limits: Boolean ;
 Procedure Print ( Line : String ) is
   Delay_Limit  : constant Duration := 1 * Seconds;
 Brgin
   Select
              Printer.The_Printer.Put_Line(Line);
              Or
              delay Delay_Limit;
              Alarme.The_Alarm.Report_printer_Error;
   End select;
  End Print;
  Use Calender;
  Begin  --- Monitor
              Loop
               Select
                      Accept Shut_down ;
               Exit;
   Or
               Delay Next_Time – Calender.Clock;
               Print("");
               For Index in Sensor_Names
                      Loop
```

```
                    The_Sensors(Idex).Get_Status(The_Value,
Out_Of_Limits);
                         Print ( Format ( Index, The_Value, Out_Of_limits )
) ;
                        End loop;
                      Next_Time ;= Next_Time + Time_Inerval;
                    end select;
                   end loop;
end Monitor;
```

لنستخدم الآن مصفوفة الرسالة التي تساعدنا في عمل المخرج. نبدؤها برسالة تحدد فيما إذا كانت القيم موجودة في الحدود أو في خارجها. يمكننا اختيار ذلك مباشرة بتأشير المصفوفة بالقيمة Out_Of_Limits .

لم يبق الآن سوى الإجرائية Process_Commands التي تدير كل التفاعلات مع المستخدم. نسمح هنا للمستخدم بعدة أوامر: إعطاء حدود لبعض المجسات، إظهار تقرير حالة خروج. تستخدم هذه الإجرائية حلقة بسيطة تخرج منه عندما يدخل المستخدم Quit.في الإجرائية التالية نستخدم تقنية البقاء في حلقة لحين يدخل المستخدم قيمة مناسبة:

```
with Text_IO
 separate (Monitor_Temperatures)
 procedure Process_Commands is
  type Command is ( Limit, Status, Quit);
  package Command_IO is new
            Text_IO.Enumeration_IO(Command) ;
   package Name_IO is new
            Text_IO.Enumeration_IO(Sensor_Names);
   package Value_IO is new Text_IO.Integer_IO(Value);
 The_Command : Command ;
   The_Name     : Sensor_Names ;
   The_Value    : Value ;
   Out_Of_Limits : Boolean ;
   procedure Complain is
   begin
```

```
    Text_IO.Skip_Line ;
    Text_IO.Put_Line("invalid command try again");
end Complain ;
     procedure Get ( The_Command : out  Command ) is
   begin     --   Get
    loop
     begin
     Text_IO.New_Line ;
     Text_IO.Put("Enter a command(limit, status,quit):" );
       Command_IO.Get ( The_Command ) ;
     return ;
      exception
       when  Text_IO.Data_Error    =>  Complain;
      end ;
    end loop ;
   end Get ;
   procedure Get ( The_Name  : out  Sensor_Names ) is
   begin   -- Get
    loop
     begin
      Text_Io.Put_Line("Possible sensor name are :  ) ;
       for Index in Sensor_Names
        loop
          Text_IO.Put(' ') ;
          Name_Io.Put(Index) ;
          Text_IO.New_Line ;
        end loop;
       Text_IO.Put("Enter  a  name :  ") ;
       Name_IO.Get(The_Name)  ;
       Text_IO.Skip_Line ;
       return  ;
      exception
       when Text_IO.Data_Error   =>  Complain  ;
      end  ;
    end loop;
  end Get ;
```

```
procedure Get ( The_Value  : out Value  ) is
begin
 Text_IO.Put"Enter  a  value  )  ;
  Value_IO.Get ( The_Value ) ;
 Text_IO.Skip_Line ;
  exception
 when Text_IO.Data_Error    =>   Complain;
   Get ( The_Value) ;
end Get ;
begin      --      Process_Commands
 loop
   Get  (  The_Command  )  ;
   case The_Command is
     when Limit  =>
     Get (The_Name)  ;
     Get (THe_Value )  ;
     The_SensorsThe_Name).Set_Limit (The_Value) ;
   when  Status  =>
     for Index in Sensor_Names
       loop
         The_Sensors(Index).Get_Status(The_Value,
Out_Of_Limits) ;
         Text_IO.Put_Line(Format(Index,The_Value,
 Out_Of_Limits ));
       end loop  ;
   when Quit   =>  exit ;
   end case ;
  end loop ;
end Process_Command  ;
```

❁ ❁ ❁

18

الدخل/الخرج
Input/Output

إدارة الملف

الدخل/الخرج للمعطيات غير النصية

الدخل/الخرج للمعطيات النصية

إن الهدف من استخدام إمكانيات الدخل / الخرج Input/Output، هو إمكانية بناء اجرائيات الدخل / الخرج I/O للإتصال مع المحيطات الخاصة. إضافة إلى ذلك، وبدون إضافة تعليمات جديدة إلى اللغة، نحتاج إلى وحدات معرفة مسبقاً لـ I/O من أنواع المعطيات العادية، مثل المحارف، والأعداد الصحيحة، والأعداد الحقيقية، والتي يمكن اختيارها حسب الحاجة. وفي هذا الفصل سنرى أن ADA تقدم الدخل والخرج وتعليمات أخرى.

١٨ - ١ - إدارة الملف (File Management) :

تزود لغة ADA عدة حزم برمجية معرّفة مسبقاً من أجل الدخل/الخرج، وهذه الحزم هي التالية:

- Sequential_IO : وهي حزمة برمجية من أجل الوصول التسلسلي للمعطيات غير النصية.
- Direct_IO : وهي حزمة برمجية من أجل الوصول المباشر (العشوائي) للمعطيات غير النصية.
- Text_IO : وهي حزمة برمجية من أجل المعطيات النصية (المحرفية).

وبالإضافة للحزم البرمجية الثلاث الآنفة الذكر، فإنّ الحزمة البرمجية IO_Exceptions، تعرّف الإستثناءات المحتاجة من قبل تلك الحزم البرمجية. وفي قسم لاحق، سوف نذكر توصيف الحزم البرمجية الخاصة بالدخل/الخرج.

وتذكّر دائماً، بأنّه في هذا الفصل، سوف نقدم إمكانيات الدخل/خرج الخاصة بلغة ADA. إذ يمكن القول، بأنّ الحزم البرمجية للدخل/خرج لا تنتج مجموعة عالية من اللغة، بالرغم من وجود الإمكانيات الجديدة في سياق وجود لغة ADA.

وإن الإجرائيات الخاصة بالدخل/خرج من أجل الحزم البرمجية الثلاث الخاصة بالدخل/خرج، تتضمن ما يلي:

- Close : من أجل إغلاق ملف خارجي.
- Create : من أجل خلق ملف خارجي جديد.

- Delete : من أجل حذف ملف خارجي.
- Open : من أجل فتح ملف خارجي موجود.
- Reset : من أجل البدء مرة ثانية من بداية الملف الخارجي.

وأيضاً، إن التوابع الفرعية التالية، متضمنة في الحزم البرمجية الثلاث الخاصة بالدخل/الخرج:

- End_Of_File: تعيد هذه الوظيفة، القيمة المنطقية:
- True : إذا لم يكن بالإمكان قراءة معلومات جديدة من الملف الخارجي.
- False : إذا مازال هنالك معلومات يمكن قراءتها في الملف الخارجي.
- Form : تعيد هذا الوظيفة شكل سلسلة المحارف للملف الخارجي.
- Is_Open : يعيد هذا الوظيفة القيمة المنطقية:
- True : إذا كان الملف الخارجي مفتوحاً.
- False : إذا كان الملف الخارجي مغلقاً.
- Mode : تعيد هذه الوظيفة نموذج الملف، إذ يأخذ أحد القيم التالية:
- In_File : للإشارة بأنّه ملف دخل.
- Out_File : للإشارة بأنه ملف خرج.
- InOut_File : للإشارة بأنّه ملف دخل وخرج بنفس الوقت.
- Name : تعيد هذه الوظيفة سلسلة من المحارف، تمثل إسم الملف الخارجي.

وهناك تفصيلات أكثر، سيتم تحديدها من خلال عرض الفصل.

١٨ - ٢ - الدخل/الخرج للمعطيات غير النصية:

(Input/Output for Nontextual Data):

إن الحزمتين البرمجيتين المولّدتين Sequential_IO وDirect_IO، تزودان جميع الأساسيات المحتاجة في الدخل/الخرج، من أجل نوع عنصر معين. وتستخدم هاتان الحزمتان البرمجيتان، من أجل المعطيات غير النصية. وبمعنى آخر، إنّ الحزمة البرمجية Sequential_IO، والحزمة البرمجية Direct_IO، تعملان حسب تمثيل

المعطيات في الآلة. وكلا الحزمتان تزودان إمكانيات افتراضية متطابقة، بينما يتجلى الفرق بأنّ الحزمة البرمجية Sequential_IO، ملائمة للملفات ذات الوصول التسلسلي، بينما الحزمة البرمجية Direct_IO، ملائمة للملفات ذات الوصول العشوائي (المباشر). والحزمة البرمجية Text_IO، تزود إمكانيات الدخل/الخرج من أجل المعطيات النصية (المحرفية).

وإن كلا الحزمتين البرمجيتين الخاصتين بالمعطيات غير النصية، تملكان الإجرائيتين Read وWrite، بينما الحزمة البرمجية Text_IO، فتملك الإجرائيتين Get وPut. وبالإضافة لذلك، فإن الحزمة البرمجية Direct_IO، تملك الإجرائية Set_Index، والتابعين الفرعيين Size وindex، للسماح للمستثمر بالوصول المباشر لعنصر خاص.

وبما أنّ كلا الحزمتين البرمجيتين مولّدتين، لذلك يجب نسخهما مؤقتاً من أجل نوع معطيات محدد، والذي يتمثل بالمعامل Element_Type.

مثال: نريد أنّ ندخل أو نخرج عدة أنواع رقمية. والبرنامج التالي يمثل ذلك:

```
With Direct_IO, Sequential_IO;
Procedure Main is
 type Dollar is delta 0.01 range 0.0..1_000.0;
 type Pyroll is record
          Name : String(1..20);
          Ssn  : Long_Integer;
          Pay  : Dollar;
          end record;
 type Part is record
        Description    : String(1..20);
        Stock_Number : Positive;
        Quantity        : Integer;
        Price            : Dollar;
        end record;
package Payroll_IO is new
    Sequential_IO (Element_Type => Payroll);
use Payroll_IO;
```

```
package Float_IO is new
 Sequential_IO (Element_Type =>Float);
use Float_IO;
package Part_IO is new
  Direct_IO (Element_Type => Part);
use Part_IO;
Float_File     : Float_IO.File_Type;
Payroll_File : Payroll_IO.File_Type;
My_Pay        : Payroll;
Average        : Float;
      Begin  -- Main
   Open (File => Payroll_File, Mode => In_File,
   Name => "Company.Pay");
        Open (File => Float_File, Mode => Out_File,
          Name => "Real_Stuff");
          .....
          Read (File => Payroll_File, Item => My_Pay);
        Write (File => Float_File, Item => Average + 25.0);
        .....
End Main;
```

وبما أنّ الحزم البرمجية الثلاث الخاصة بالدخل/خرج تمثل وحدات مكتبية، لذلك، يجب إستيرادها باستخدام عبارة With. لاحظ أننا استخدمنا مجموعة المعاملات المسمّاة، وذلك، لجعل النسخة المؤقتة مقروءةً بشكل أفضل. ولاحظ أنّه بإمكاننا خلق حزمة برمجية مؤقتة للدخل/الخرج، لأنواع المعطيات المعرّفة مسبقاً مثل Integer وFloat. وإن أنواع المعطيات المعرفة من قبل المستخدم والمركبة (التسجيلات، والمصفوفات مثل Payroll و Part)، يمكن استخدامها من أجل الملفات غير النصية، مثلما تمّ تقيدها سابقاً. بالإضافة لذلك، فإن تنفيذ (زرع) برنامج، يمكن أن يمنع إجراء نسخ مؤقتة من الحزمتين البرمجيتين الخاصتين بالمعطيات غير النصية، من أجل أنواع الوصول.

وكنتيجة طبيعية لقواعد لغة ADA القوية، يجب إجراء نسخة مؤقتة عن حزمة برمجية، من أجل كل نوع معطيات تريد إدخاله أو إخراجه. وقد تمّ تصميم لغة ADA

بمفهوم أن نكتب الترميز (Code) مرةً واحدةً فقط، وقراءته عدة مرات وحسب الحاجـة. وهذه حالة واحدة كتبنا فيها الترميز ليصبح طويلاً، والذي مازال فيها صريحـاً، إذ تـمّ استخدام مجموعة من الحزم البرمجية الخاصة بالدخل/الخرج في تطبيق خاص، يمكن استنتاج أنّك صرّحت عن النسخة المطابقة بأنها وحدة مكتبية، أو تمّ وضعها فـي قسم توصيف الحزمة البرمجية المكتبية. وهذه الطريقة، تقلّص ترويسة أي دخل/خرج يمكن أن تتواجد في نظام محشور، بتأكيد أننا سرّعنا فقط إجرائيـات الدخـل/الخـرج الثابتـة المحتاجـة فعليـاً. وعلـى سـبيل المثـال، فيمـا يلـي تصريـح عـن حزمتيـن برمجيتيــن مكتبيتيــن، تــمّ نســخهما مؤقتــاً عــن حزمتيــن برمجيتيــن مولّدتيــن، خـــاصتين بالدخل/الخرج، ومعرّفتين مسبقاً:

```
With Sequential_IO
package Integer_IO is new
      Sequential_IO(Element_Type => Integer);
With Measures, Text_IO;
package Kg_IO is new
   Text_IO.Integer_IO(Measures.Kilograms);
```

لاحظ أيضاً، بأنّ جميـع الحـزم البرمجيـة الخاصـة بـالدخل/الخـرج، والمعرّفـة مسبقاً، هي تسلسلية؛ وبأنّ علمها الدلالي مؤهل فقط، في حال وجود مهمـة وحيـدة. بينما في حال وجود عدة مهمات تعالج الملفات، فمن المهم أن تكون جميع العمليـات تسلسلية، من خلال آلية تعريف بعض المبرمجين.

بنية الملف (File Structure) :

إن جميع المستويات العالية للدخل/الخرج فـي لغـة ADA مرتبطـة مـع ملـف. ويمثل الملف سلسلةً منتهيةً من العناصر. وجميع عناصر الملف، يجب أن تكـون مـن نوعٍ واحد. وخارجياً، فـالملف مرتبـط بـأداة فيزيائيـة، مثـل القرص، أو الطرفـي، أو الطابعة. وداخلياً، فإن جميع عناصر الدخل/الخـرج، تعـالج منطقيـاً مـن خـلال ملـف غرض. وكل عملية في ملف، معرّفة على أغراض لنوع ملف محدد. ومن المثـال Main المحدد سابقاً، يمكن التصريح عن الملفات التالية:

```
with Text_IO; use Text_IO;
with Sequential_IO;
procedure Main is
  type Dollar is Delta 0.01 range 0.0..1_000.0;
  package Integer_IO is new
            Integer_IO (Num => Integer);
  package Float_IO is new
          Sequential_IO (Element_Type => Float);
  package Dollar_IO is new
           Fixed_IO (Num => Dollar);
  Integer_File : Text_IO.File_Type;
  Float_File    : Float_IO.File_Type;
  Dollar_File   : Text_IO.File_Type;
Begin
  ....
End Main;
```

وقد تمّ تحديد نموذج الملف، عندما تمّ فتح الملف أو خلقه. وفيما يلي، النماذج الثلاثة للملفات التي تمّ تعريفها:

In_File
Out_File
InOut_File

والنموذجان In_File وOut_File، صالحان من أجل الحزم البرمجية الثلاثة الخاصة بالدخل/الخرج. بينما النموذج InOut_File، فصالح فقط، من أجل الملفات ذات الوصول المباشر. وإسم كل نموذج، يحدد جهة تدفق المعطيات بالنسبة للبرنامج. فعلى سبيل المثال، فملفٌ من النموذج In_File، يمكن استخدامه فقط، من أجل معطيات الدخل. ونموذج الملف يمكن أن يتغير في زمن التنفيذ، وذلك لحظة فتحه، باستخدام الإجرائية Reset. ومن الجدير بالذكر هنا، أنّه ليس جميع البيئات البرمجية تستخدم Reset، لأن ذلك يقلص إمكانية النقل.

والملف ذو الوصول المباشر DirectIO، يُرى بأنّ مجموعة من العناصر (جميعها من نفس النوع) تحجز مواضع متتالية، وفق ترتيب خطي. وهذا لا يشبه المصفوفة أحادية البعد. ويمكننا نقل قيمة من أو إلى عنصر من الملف، وفي أيّ موضع مختار.

وأي ملف له حجم حالي محدد، يشير إلى عدد العناصر المتواجدة في الملف. ومواضع الملف مفهرسة من العدد ١، وتنتهي بحجم الملف الحالي. بالإضافة لذلك، تعرِّف لغة ADA قيمةً طبيعيةً تسمى Current_Index، لتمثل الموضع الحالي الذي سيُقرأ منه أو سيكتب فيه؛ والتابع Index، يعيد هذه القيمة. ويمكن أن نعطي قيمةً لـ Current_Index بصراحة، وذلك باستخدام الإجرائية Set_Index.

ويفضل استخدام الملفات ذات الوصول التسلسلي، بدلاً من الملفات ذات الوصول المباشر، في التطبيقات التي لا تتطلب الوصول المباشر. والسبب في ذلك، هو أنّ الحزمة البرمجية Sequential_IO، تعطي تجريداً أفضل. ومن أجل الملفات ذات الوصول التسلسلي، لا يوجد مفهوم اختيار الموضع، وبالتالي، يتم نقل القيم بسهولة، وفق ترتيب ظهورها. والإجرائية Reset، يمكن أن تغير الموضع من أي مكان في الملف التسلسلي إلى بدايته.

معالجة الملف (File Processing):

من أجل التماسك، فإن إجرائيات معالجة الملف في لغة ADA، مستقلة عن أيّة حدود فيزيائية. وكنتيجة لذلك، فإن بعض الإجرائيات التي سنصفها، ليست مطابقةً من أجل جميع الملفات الفيزيائية، إذ لا يمكننا الكتابة في ملف، لا يسمح نظام الإستثمار لأيّ أحدٍ بالوصول إليه. فإذا حاولنا تطبيق عملية غير مطابقة لملف، عندها سيقدم النظام إستثناءات. والحزمة البرمجية IO_Exceptions، تعرِّف عدة إستثناءات مطابقة لمعالجة ملف. ومن أجل التلاؤم، فقد أُعيدت تسمية هذه الإستثناءات، في قسم توصيف الحزم البرمجية Text_IO, Direct_IO, Sequential_IO.

وفيما يلي، لائحة بالإستثناءات، وأسباب ظهورها:

- **Data_Error** : يظهر هذا الإستثناء، إذا لم يتطابق نوع ناتج عملية الدخل، مع النوع المتوقع، مثلاً، عندما يحاول برنامج ما، أن يقرأ قيمةً صحيحة، لكنه يستقبل محرفاً بدلاً منها. أو إذا تمّت قراءة القيمة -٢٣ من ملف، قد تمّ تعريف معطياته بأنها أعداد طبيعية موجبة.

- Device_Error : يظهر هذا الإستثناء، إذا كان هنالك خلل في النظام.
- End_Error: يظهر هذا الإستثناء، إذا حاولنا القراءة بعد نهاية الملف.
- Mode_Error : يظهر هذا الإستثناء، إذا حاولنا القراءة من ملف له النموذج Out_File، أو حاولنا الكتابة في ملف له النموذج In_File.
- Name_Error : يظهر هذا الإستثناء، إذا حاولنا خلق أو فتح ملف ممنوع، أو أن إسمه غير وحيد.
- Status_Error : يظهر هذا الإستثناء، إذا حاولنا الكتابة أو القراءة من ملف غير مفتوح، أو إذا حاولنا فتح ملف مفتوح من قبل (لم يُغلق بعد).
- Use_Error : يظهر هذا الإستثناء، إذا حاولنا تطبيق عملية غير مسموح بها على الملف الفيزيائي المعرّف. ومثال ذلك، إذا كتبنا على ملف لا يسمح نظام الإستثمار بالولوج إليه، أو بالكتابة في ملف متواجد على قرص ممتلئ.

لاحظ أنّ فحص شروط الـ Data_Error، إختياري بالنسبة لـ Sequential_IO وdirect_IO. وأن بالنسبة للحزمة البرمجية Text_IO، وبالإضافة إلى الإستثناءات الآنفة الذكر، فيوجد الإستثناء Layout_Error، والذي يعود فقط لمعالجة الـ Text_IO.

وقبل بدء معالجة أيّ ملف، يجب ربط ملف غرض بملف فيزيائي. وإن البرامج الجزئية Create وopen، أُستخدِمت لإجراء هذا الربط. بالإضافة لذلك، فإن هذه البرامج الجزئية، تسمح لنا بتسمية نموذج الملف. ونستخدم Create لبناء ملف جديد، بينما نستخدم Open، من أجل فتح ملفات موجودة مسبقاً. ويتم استدعاء هذه الإجرائيات، بالشكل التالي:

```
Create (File    => Payroll_File,
        Mode => Out_File,
        Name => "SystemFile",
        Form => "Disk2");
Open (File => Part_File, Mode => In_File,
                    Name => BlackBoxFile);
```

ومرة ثانية، قد استخدمنا طريقة المعاملات المسمّاة وذلك لجعل الإستدعاء أكثر قابليةً للقراءة. إذ أنّ:

- File : يشير إلى ملف الغرض، ليتم ربطه.
- Mode : يعرِّف نموذج الملف المنطقي، ويأخذ إحدى القيم In_File, Out_File، InOut_File.
- Name : وهو عبارة عن سلسلة محارف، تعرِّف ملفاً فيزيائياً خارجياً. وإسماء الملفات الخارجية، تتعلق بالتنفيذ، ومؤسسةٌ على اصطلاح التسميات المستخدمة في نظام مُعطى.
- Form : سلسلة محارف إختيارية، ويتعلق أيضاً بالتنفيذ. وإن مختلف النظم تستخدم Form، لتسمح للمستخدم بتعريف عدة خواص للملف، مثلاً "الحفظ كل ٣٠ يوم".

لاحظ أنّه من أجل الـ Name والـ Form، لا يمكننا استخدام سلسلة محارف حرفية، لكن يمكننا استخدام سلسلة محارف غرضية، ومثال ذلك ما يلي:

```
With Text_IO; Use Text_IO;
Procedure Main is
 Float_File : File_Type;
 Last       : Natural;
 Title      : String(1..80);
Begin
 Get_Line(Title,Last);--Read in a title from the user
 Create(File => Float_File, Name => Title(1..Last));
End Main;
```

لاحظ أيضاً، من أجل الملفات التسلسلية والنصية المخلوقة حديثاً، فإن النموذج البدائي هو دائماً Out_File، بينما من أجل الملفات ذات الوصول المباشر، فإنّ النموذج البدائي هو دائماً InOut_File. وبالإضافة لذلك، إذا كانت القيمة الإفتراضية لإسم الملف المخلوق حديثاً، تمثل سلسلةً فارغة، وعندها، يخلق النظام ملفاً مؤقتاً إسمه لا يهمنا، وهذا الملف، سوف ينتهي عند انتهاء البرنامج.

وبعد أن أنهينا معالجة الملف، يجب أن نغلقه بشكل صريح، وفق إحدى الطريقتين التاليتين:

```
Close(Part_File);
Delete(Float_File);
```

ففي المثال الأول، نلغي بسهولة، الإرتباط بين الملف الغرض وإسم النظام الخارجي، ويُحتفظ بالملف الفيزيائي في مكان تواجده. بينما في المثال الثاني، نلغي الإرتباط بين الملف الغرض وإسم النظام الخارجي، إذ يتم حذف الملف الفيزيائي من مكان تواجده.

وتزود لغة ADA خمس توابع أساسية لفحص حالة الملفات. وجميع هذه التوابع، تتطلب معامل دخل وحيد، وهو الملف الغرض. وهذه التوابع ما يلي:

- **Name** : يعيد سلسلةً من المحارف، تمثل الإسم الفيزيائي للملف.
- **Form** : تعيد سلسلة محارف، تمثل الشكل الحالي للملف المحدد.
- **Is_Open**: تعيد القيمة المنطقية True، إذا كان الملف مفتوحاً، وإلا فتعيد القيمة False.
- **End_Of_File** : تعيد القيمة المنطقية True، إذا وصلنا لنهاية الملف، ولم نستطع قراءة أي عنصر آخر، وتعيد القيمة False في غير ذلك.
- **Mode**: تعيد النموذج الحالي للملف المحدد.

أما بالنسبة للملفات ذات الوصول المباشر، فتعرِّف ADA أيضاً، التوابع التالية:

- **Size** : يعيد عدد العناصر الحالية في الملف.
- **Index** : يعيد قيمة الموضع الحالي، الذي سنكتب فيه أو نقرأ منه.

ومن أجل الملفات ذات الوصول المباشر فقط، يمكننا وبشكل صريح، إعطاء قيمة للموضع الحالي الذي سنكتب فيه أو نقرأ منه، وذلك باستخدام Set_Index. ومثال ذلك ما يلي:

```
Open (File => Part_File, Mode => InOut_File,
               Name => Inventory.Part");
 .....
Set_Index(Part_File, To => 137);
```

وفي هذه النقطة، فإن التابع Index سيعيد القيمة ١٣٧. وإذا حاولنا إعطاء قيمة للموضع الحالي أكبر من حجم الملف، فلن يكون هنالك أي إستثناء إلا إذا حاولنا القراءة من الموضع غير المعرَّف. وعند تشكيل ملف خرج، فإنه يمكننا إعطاء قيمة الموضع الحالي لما بعد نهاية الملف، ونكتب العنصر الجديد.

ومن أجل الملفات التسلسلية والملفات ذات الوصول المباشر، فتتم معالجة أرشفة الملفات، بالتابعين الأساسيين Read وWrite. وبالتعريف، فإنه يمكننا القراءة من ملف، إذا كان من النموذج In_File أو من النموذج InOut_File. كما أنه يمكننا الكتابة في ملف، إذا كان من النموذج Out_File، أو من النموذج InOut_File. ومرة ثانية، يجب أن نتذكر، بأن النموذج InOut_File لا يوجد إلا في الملفات ذات الوصول المباشر. وفي أية حالة، يجب أن نعطي الملف المنطقي، بالإضافة لقيمة أو غرض من نوع العنصر المطابق. ومثال ذلك ما يلي:

```
Read(File => Payroll_File, Item => My_Pay);
Write(File => Float_File, Item => Average+25.0);
```

وفي أية لحظة يتم بها استدعاء هذين التابعين، فإن التأثير، هو إدخال أو إخراج القيمة المعطية بالشكل الثنائي، وفق التمثيل الداخلي للآلة. فإذا كان الارتباط مع أداة دخل/خرج وحيدة، فإنه يمكننا استخدام تمثيل التوصيفات، لصياغة شكل القيم كما تم نقلها أو استقبالها من الجهاز. وسنضيف أيضاً، بأنّ الحزمة البرمجية Direct_IO، تصدر شكلاً من البرامج الجزئية Read وWrite مع المعامل Index، الذي يعرِّف صراحةً المكان الذي سيتم منه القراءة أو الكتابة. أيضاً Read وWrite في الملفات ذات الوصول المباشر، تزيد آلياً موضع الملف بعد انتهاء عملية الدخل/الخرج. وأخيراً، لاحظ أنّ التمثيل الثنائي لنوع، يمكن أن يختلف من نظام لآخر. إذاً، على

نظام واحد، قد استخدمنا النسخة التالية من الـ Sequential_IO، للكتابة في ملف تسلسلي:

```
With Sequential_IO;
procedure Main is
 type Integer_16 is range -32_768..32_767;
 for Integer_16'Size use 16;
 Package Integer_16_IO is new Sequential_IO(Integer_16);
Begin
  ...
End Main;
```

وبعد ذلك استخدمنا نفس الترميز من أجل قراءة الملف من نظام آخر، حيث أن الأعداد التي كتبت، ربما تختلف عن الأعداد التي قرئت، فمثلاً، إن تمثيل القيمة الصحيحة ١ في أحد الأنظمة، يمكن أن يكون على الشكل 1000000000000000، بينما في نظام ثانٍ يمكن أن يكون على الشكل 1000000000000000.

ولإثبات تسهيلات الدخل/الخرج عالية المستوى، والتي تزودها لغة ADA، يمكننا أن نرى ذلك من خلال المثال التالي.

مثال: نقرأ في هذا المثال، سلسلةً من الأعداد الصحيحة من ملف، ونحسب المجموع الكلي، ومن ثم، نكتب ناتج الجمع في ملف نخلقه حديثاً. والبرنامج التالي يمثل ذلك:

```
With Sequential_IO;
Procedure Simple_Example is
  Package Integer_IO is new
     Sequential_(Element_Type    => Integer);
  Input_Data    : Integer_IO.File_Type;
  Output_Data : Integer_IO.File_Type;
  Value           : Integer;
  Sum             : Integer :=0;
  use Integer_IO;
Begin   -- Simple_Example
  Open  (Input_Data, In_File, "C:My_Input.dat");
  Create (Output_Data, Out_File, "A:My_Output.dat");
```

```
 Loop
   exit when End_Of_File(Input_Data);
   Read(Input_Data, Value);
   Sum := Sum + Value;
 End Loop;
 Write (Output_Data, Sum);
 Close(Input_Data);
 Close(Output_Data);
End Simple_Example;
```

وبما أنّ حجم الملف غير معروف، لاحظ كيف تمّ استخدام التابع End_Of_File، للسيطرة على عدد القراءات من الملف.

١٨ - ٣ - الدخل/الخرج للمعطيات النصية

Input / output for Textual Data :

باستخدام تسهيلات الحزمة البرمجية Sequential_IO، وتسهيلات الحزمة البرمجية Direct_IO، يمكننا نظرياً أن نحصل على جميع العمليات الأساسية المحتاجة لتشكيل الدخل/الخرج لأي نوع معطيات. ويمكننا إيجاد نسخة عن حزم برمجية للتعامل مع العناصر المحرفية، ولكننا نريد أن نبني برامجنا الخاصة، للحصول على الدخل/خرج الخاص بالأعداد، لتصبح بشكل مقروء بالنسبة للمستثمر. ومن أجل هذه الغاية، عرّفت لغة ADA حزمةً برمجيةً منفصلة، ليست مولّدة، تُدعى بـ Text_IO. وتزود هذه الحزمة البرمجية توابع شبيهة بتلك الخاصة بالحزم البرمجية المولّدة. وبالإضافة لذلك، تزود ببرامج جزئية صالحة لتشكيل النصوص.

بنية الملف (File structure):

تزود الحزمة البرمجية Text_IO تسهيلات خاصة بالدخل/الخرج، من أجل معطيات الدخل/الخرج المكونة من محارف الـ ASCII. فقط، شبيهاً للحزم البرمجية المولّدة، فإن معالجة جميع الـ Text_IO، تتم من خلال ملفات. بالإضافة لذلك، فعند استخدام الحزمة البرمجية Text_IO، يتم فتح ملفات الدخل/الخرج المعيارية (Standard) في بداية تنفيذ البرنامج، والتي تتمثل بلوحة المفاتيح، وشاشة الحاسوب.

فإذا لم يحدد ملف الدخل، فإنّ لوحة المفاتيح تمثله. أيضاً، إذا لم يحدد ملف الخرج، فإنّ الشاشة تمثله. وتسمح عدة نظم للمستخدم بتحديد الملفات التي سترتبط بملفات الدخل/الخرج المعيارية عند تنفيذ البرنامج. وتذكر أيضاً، بأنّ الحزمة البرمجية Text_IO، غير صالحة للعمل في حال وجود عدة مهمات (Multiple Tasks). ووفقاً لهذا، في تطبيق ما، إذا وجدت مهمة واحدة تُدخل المعطيات ومهمة واحدة تخرج المعطيات، فإنّ تصرف البرنامج سيتغير أثناء التنفيذ.

وبالطبع، يمكننا خلق وفتح ملفات نصية خاصة بشكل صريح. والمثال التالي، يوضح ذلك:

```
With Text_IO;
Procedure Main is
  My_Input, My_Output : Text_IO.File_Type;
Begin
 Text_IO.Open(My_Input, Text_IO.In_File, "Data_Set_I");
 Text_IO.Creat(My_Output, Text_IO.Out_File, "Output");
  .....
End Main;
```

لاحظ أنه، وبما أنّ Text_IO تمثل وحدةً مكتبية، لذلك، يجب استيرادها باستخدام العبارة With، كما هو الحال في Sequential_IO، وفقط، الملفات ذات النموذج In_File والنموذج Out_File يمكن فتحها؛ إذ أنّ الملفات ذات النموذج InOut_File، لا يمكن التعامل معها في الملفات النصية. وبشكل مشابه للحزم البرمجية المولّدة والخاصة بالدخل/الخرج، فإنّ Text_IO، تملك جميع إجرائيات الدخل/الخرج، مثل Close, Create, Delete, Open, Reset، والتي تم التنويه عنها من قبل.

وإن جميع العمليات الخاصة بـ Text_IO، قد تمّ تعريفها كغرض محدد فيه الـ File_Type. وكنتيجة للسماح بملفات غرض بدائية، فإن معظم البرامج الجزئية في الـ Text_IO لها شكلان للمعاملات: الأول، لا يتضمن أيّ ملف، ويستثمر الملفات البدائية؛ بينما الثاني، يتطلب التصريح عن ملف الغرض. وفي البدء، فإن الملفات

البدائيـــة، تتمثّـل بالملفـــات المعياريـــة. وباســـتخدام التـــابعين Standard_Input وStandard_Output، يمكننا كسب قيمة الملفات المعيارية. بالإضافة لذلك، فإن التابعين Set_Input وSet_Output، يسمحان لنا بإعادة ربط الملفات البدائية أثناء زمن التنفيذ، بينما التابعين Current_Input وCurrent_Output، فيعيدان الملفات البدائيـة. والمثال التالي، يوضح ذلك:

```
With Text_IO;
Procedure Redirection is
  Log : Text_IO.File_Type;
  Log_Name : String(1..100);
  Name_Last : Natural;
  Use Text_IO;
Begin
 Put("Enter a log file name: ");
 Get_Line(Log_Name, Name_Last);
 Create (Log, Out_File, Log_Name(1..Name_Last));
 Set_Output(Log);
 Put_Line("Hello.. This is a log.");
 Set_Output(Standard_Output);
 Close(Log);
 Put_Line("Done.");
End Redirection;
```

وإن التعليمة ;("Put("Enter a log file name: تؤدي إلى كتابة Enter a log file name: على الخرج المعياري الممثل بالشاشـة. وبعد خلق الملف log، فقد جعلناه الخرج البدائي. وبعد التعليمة Set_Output(Log)، فإنّ التابع Current_Output سيُعيد Log . وعند تنفيذ التعليمة Put_Line("Hello.. This is a log.")؛ فإنّ الجملة Hello.. This is a log. ستكتب في الملف Log الذي تمّ خلقه، دون أن تظهر على الشاشة. أما تنفيذ التعليمة Set_Output(Standard_Output) فسيغير ملف الخرج البدائي ليظهر من جديد على الشاشة. أما عند تنفيذ التعليمة ;("Put_Line("Done. فستظهر الرسالة Done. على الشاشة، دون أن تخزّن في الملف Log.

ومن أجل الملفات النصية، فإن التوابع Is_Open, End_Of_File, Mode, Name, Form تكون صالحة أيضاً. وهذه التوابع، تعمل بنفس الطريقة التي وصفت بها، من أجل الحزم البرمجية المولّدة الخاصة بالدخل/الخرج.

تنظيم الملفات (File Layout):

تزود Text_IO بموجهات أسطر للخرج، وتجعل الخرج مقروءاً بشكل جيد، عند توجيهه على بعض الآلات مثل الطابعة أو شاشة العرض، أو على أقراص. وتزود لغة ADA بإمكانيات تساعد بتوصيف الخرج. والملف النصي، يمكن أن يكون Standard_Output أو ملف مخزن على قرص، يمثل كصفحات ذات أبعاد X سطر وY محرف. وتتألف كل صفحة من لائحة من الأسطر. وعدد أسطر الصفحة، يمكن أن يكون متغيراً أو ثابتاً، ويتألف كل سطر من عدة أعمدة ذات طول ثابت أو متغير. وتزودنا الـ Text_IO بالصفحة الحالية، والسطر ضمن الصفحة الحالية، وموضع العمود في الملف.

وإن طول السطر وطول الصفحة لملف من النموذج Out_File، يمكن أن يعين باستخدام الإجرائيات Set_Line_Length وSet_Page_Length. فعلى سبيل المثال:

Set_Page_Length(My_Output, To => 66);

وأما تفصيل التوصيف للإجرائية Set_Page_Length، وإجرائيات أخرى، فستتم مناقشته بالتفصيل فيما بعد.

إذا حاولنا إخراج أكثر من 66 سطراً، فإن خاتم الصفحة (يدعى في بعض الأحيان بـ "Page Break"، أو "Form Feed") سيتم إخراجه آلياً بعد السطر ذي الرقم 66 . ولتعيين خاتم الصفحة بشكل صريح، ننجز أحداً مما يلي:

```
New_Line;  ---  Output a line terminator
New_Page;  --- Output a page terminator
```

وعندما نريد أن نُخرج صراحة خاتم سطر، أو خاتم صفحة، وليس آلياً يتم هذا الإخراج بإعطاء القيمة صفر للطول. وفي Text_IO، فإن الطول صفر، يعني طولاً غير

محـــدود (Unbounded). ويمكـــن أن نتســـاءل عـــن المعــــاملات الحاليـــــة للتـــابعين Line_Length وPage_Length .

وإن جـزءاً مـن الإجرائيـات Put وget، والتـي تؤثـر مضمـراً علـى قيمـة العمـود الحالي، هو أنه يمكن إعطاء قيمة بدائية صريحة لعداد الأعمدة. وإن Text_IO، تعرّف برنامجين جزئيين مرتبطين بموضع العمود:

- **Col** : يعيد قيمة العمود الحالية.
- **Set_Col** : يعطي العمود الحالي.

فإذا حاولنا إعطاء قيمة لموضع العمود أكبر من طـول السـطر الحـالي، فسـيظهر الإستثناء Layout_Error مشيراً إلى خطأ.

وأيضاً، إن Text_IO، تزود عدة برامج جزئية ترتبط بموضع السطر الحالي:

- **Line** : يعيد قيمة السطر الحالي.
- **Set_Line** : يغير السطر الحالي.

وللإنتقال إلى سطر جديد، فإن Text_IO تزود الإجرائيات التالية:

- **New_Line**: تكتب خاتم سطر أو أكثر.
- **Skip_Line**: تقرأ نهاية سطر أو أكثر، قد انتهى.

وأيضاً، هنـالك التـابع End_Of_Line، الـذي يعيـد القيمـة المنطقيـة True إذا وصلنا لنهاية السطر الحالي، ويعيد القيمة False في غير ذلك.

وتعرّف Text_IO عدة عمليات، تؤثّر على رقم الصفحة الحاليـة. وهـذه الـبرامج الجزئية، موازية للعمليات المعرّفة على رقم السطر، بشكل خاص:

- **Page** : يعيد رقم الصفحة الحالية.
- **New_Page**: تكتب صفحة جديدة أو أكثر.
- **Skip_Page**: يقرأ خاتم صفحة أو أكثر، قد انتهت.
- **End_Of_Page**: يعيد القيمة المنطقية True، في حال الوصول إلـى نهايـة الصفحة، ويعيد القيمة False، في غير ذلك (فقـط، مـن أجـل الملفـات ذات النمـوذج In_File).

وبهذه التسهيلات، يمكننا استعراض مثال بسيط، يكتب سبعة خواتم سطر على الشاشة، ومن ثم يحدد قيمة الموضع الحالي في العمود ذي الرقم 26، كما يلي:

```
New_Line(Spacing => 7);
Set_Col( To => 26);
```

والتأثير على الشاشة، سيحرك المؤشر إلى العمود ذي الرقم 26، وبعد سبعة أسطر اعتباراً من السطر الذي يتواجد فيه المؤشر (حاول أن تكتب برنامجاً بسيطاً، تستوضح من خلاله ذلك).

لاحظ أنّ المحرف الفعلي، أو المحرف الذي يكون خاتم صفحة أو سطر، يتعلقان بالآلة.

Text_IO، من أجل الأنواع والسلاسل المحرفية

(Character and String Types):

نحصل على جميع المعاملات للملفات الأساسية في Text_IO، بالتحميل الزائد للبرامج الجزئية Put وGet. ومن أجل أنواع المحارف والسلاسل المحرفية، فإن Put، ستُخرج القيمة المعينة ابتداء من الصفحة والسطر والعمود الحالي. وإن Get، ستُدخل إلى متغير معين ابتداءاً من الصفحة والسطر والعمود الحالي. وبعد تنفيذ عملية Put أو Get، سيتم تغيير أرقام الصفحة والسطر والعمود، لتدل إلى الموضع الجديد، بعد كتابة أو قراءة القيم.

وغالباً ما نربط العمليتين Put وGet، بالعملية New_Line، والعملية Skip_Line. مثلاً، لنفترض بأنّ طول سطر 10، وموضع السطر والعمود الحالي1، فإن تنفيذ هذا يتم كما يلي:

```
with Text_IO; use Text_IO;
procedure Main is
Be
  Set_Line_Length(10);
  Set_Col(1);
  Put("Now ");
  Put("is the time ");
```

```
 put("for ");
 New_Line;
 Put("all people");
 New_Line;
End Main;
```

وعند تنفيذ هذا البرنامج البسيط، سيكون الخرج ما يلي:

```
Now is the
 time for
all people
```

وبالرغم من أننا استخدمنا تعليمتي New_Line في هذا البرنامج، لكن النتيجة تمّ عرضها على ثلاثة أسطر، والسبب في ذلك يعود إلى تحديد طول السطر بعشرة محارف، الذي أدى إلى إنهاء عرض السطر الأول عند كلمة the، والإنتقال إلى سطر جديد. لاحظ أيضاً، أنّه من أجل إخراج (و إدخال) سلاسل المحارف، فإن عرض القيمة المقروءة أو المكتوبة، يساوي طول السلسلة المحرفية.

وتزود Text_IO إجرائيتين إضافيتين للقراءة والكتابة، مرتبطتين بخاتم السطر. وهاتان الإجرائيتان هما Put_Line وGet_Line، حيث أن Put_Line، تُخرج السلسلة المحرفية المحددة على سطر، وتقفز لسطر جديد، بعد وضع خاتم سطر للمتابعة على سطر جديد. بينما Get_Line، تقرأ محارف من الموضع الحالي من السطر، وتنتهي القراءة عند الوصول إلى خاتم سطر، أو عندما يمتلئ متغير السلسة المحرفية، قبل الوصول إلى خاتم سطر. وفي حال الحصول على خاتم سطر أثناء القراءة، فإن تنفيذ Skip_Line، يؤدي لقراءة ما قبل خاتم السطر، وإن موضع العمود الحالي يأخذ القيمة1. بينما في حال ملء المتغير قبل الحصول على خاتم سطر، فلن ينفذ Skip_Line ، وإن موضع العمود الحالي يأخذ المحرف الذي يأتي مباشرة بعد المحرف الذي قرئ. وGet_Line، يعيد سلسلة محارف، وفهرساً يشير إلى موضع آخر حرف تمّت قراءته.

ولاحظ أخيراً، بأنّ قراءة وكتابة المحارف البيانية ممكنة. وتتمثل المحارف البيانية بالأحرف الأبجدية، والأرقام العشرية، ومحارف التنقيط. وإن قراءة وكتابة

محارف التحكم (إعتباراً من ASCII.Nul وحتى ASCII.Us وASCII.Del)، تتعلق بالآلة.

Text_IO، من أجل أنواع معطيات أُخرى (Other Data Types):

من أجل الملفات النصية، والتطبيقات التفاعلية، توجد حاجة لمعالجة قيم الدخل/الخرج، وتحويلها إلى أنواع رقمية، وأنواع مرقمة، وليس فقط قيم محرفية. فإذا استخدمنا الإجرائيات القياسية المتوفرة في Sequential_IO أو Direct_IO، فإن النتيجة ستكون في شكل غير نصي، وغير مقروءة بالنسبة للإنسان. ومرة ثانية، فإن لغة ADA تعرّف هذا وكأنّه حالة مشتركة، ويتضمن داخله تسهيلات الـ Text_IO من أجل الدخل/الخرج ذي الأنواع الرقمية والمرقمة. وفيما يلي أربع حزم برمجية مولّدة تم تصديرها من قبل Text_IO:

- **Integer_IO** : دخل/خرج نصي من أجل الأنواع الصحيحة.
- **Float_IO** : دخل/خرج نصي من أجل الأنواع الممثلة بالنقطة العائمة.
- **Fixed_IO** : دخل/خرج نصي من أجل الأنواع الممثلة بالنقطة الثابتة.
- **Enumeration_IO** : دخل/خرج نصي من أجل الأنواع المرقمة.

كما ذكرنا سابقاً، فقد تمّ استخدام الإجرائيتين Put وGet للحصول على هذه الأشكال من الدخل/الخرج النصي. على أي حال، وبما أنّ الأنواع الرقمية والأنواع المرقمة سيُعرّفها المستخدم، فإنّه يجب الحصول على نسخةٍ مؤقتةٍ من وحدةٍ مولّدةٍ مطابقة، والتي تعتبر جزءاً جاهزاً في الحزمة البرمجية Text_IO . ومن أجل الأنواع الرقمية، فإن أية واحدة من الحزم البرمجية الثلاثة الأولى السابقة الذكر يمكن إيجاد نسخة مؤقتة عنها، وذلك بالاعتماد على النوع الرقمي، والذي يمكن أن يكون صحيحاً، أو ممثلاً بالنقطة العائمة، أو ممثلاً بالنقطة الثابتة. وفيما يلي أمثلة عن نسخٍ مؤقتةٍ للحزم البرمجية المولّدة الأربعة، الخاصة بالدخل/الخرج:

```
With Text_IO;
procedure Main is
```

```
type Index is range 0..100;
package Index_IO is new Text_IO.Integer_IO(Index);
type Mass is digits 10;
package Mass_IO is new Text_IO.Float_IO(Mass);
type Length is delta 0.125 range 0.0..10.0;
package Length_IO is new Text_IO.Fixed_IO(Length);
package Bool_IO is new Text_IO.Enumeration_IO(Boolean);
Begin
  ....
End Main;
```

تعطي Text_IO إمكانية التحميل الزائد للبرنامج الجزئي Get للحزم البرمجية الأربعة المولّدة. وفي كل حالة، يتم الحصول على الدخل وفق التوصيف الحر. ومن أجل كل عملية لـ Get، فإن Text_IO تعرّف توصيفاً يتضمن حقلاً صريحاً، خاصاً بالعرض (Width)، ويتضمن عدد المحارف التي ستتم قراءتها. هناك عدة أشكال لـ Get، سنشرحها بالتفصيل فيما بعد. وبشكل عام، وعلى أي حال، فعندما نقرأ قيمة، يتم القفز عن الخواتم أو الفراغات، وبعدها تقرأ المحارف حتى ينهي المحرف التالي القاعدة اللغوية (Syntax) التي تشير إلى النوع أو حتى عدد الأحرف المقروءة يساوي العرض المحدد للقراءة (Width). فإن الإستثناء Data_Error يتم تنشيطه في حال كون السلسلة المقروءة لا تطابق القاعدة اللغوية المحددة (بشكل خاص، إذا لم يكن بالإمكان قراءة أي محرف، أو المحرف المقروء غير مرتبط بقيمة رقمية تمّت قراءتها عند استخدام أي من الحزم البرمجية الرقمية الخاصة بالدخل/الخرج).

وبما أنّ التمثيل النصي لقيمة رقمية ذات طول متغير، فإنّ Text_IO تعرّف البرنامج الجزئي المحمل زائداً Put. والبرنامج الجزئي Put يتضمن معاملاتٍ تخص Width وBase من أجل النوع الصحيح. وFore وAft من أجل التمثيل وفق النقطة الثابتة أو النقطة العائمة. وأيضا Put تتضمن المعامل Exp من أجل التمثيل وفق النقطة الثابتة والنقطة العائمة، والأمثلة التالية توضح ما سبق ذكره (الرمز ¶ يعني فراغ):

```
with Text_IO;
procedure Main is
 type Index is range 0..100;
 package Index_IO is new Text_IO.Integer_IO(Index);
 type Mass is digits 10;
 package Mass_IO is new Text_IO.Float_IO(Mass);
 type Length is delta 0.125 range 0.0..10.0;
 package Length_IO is new Text_IO.Fixed_IO(Length);
 package Bool_IO is new Text_IO.Enumeration_IO(Boolean);
use Text_IO;
Begin
 -- Operation                                           Display
  Index_IO.Put(26); New_Line;                           -- 26
  Index_IO.Put(26, Width =>5); New_Line;                -- ¶¶¶26
  Index_IO.Put(26, Width =>6, Base => 8); New_Line;    -- ¶8#32#
  Mass_IO.Put(3.14159, Fore=>1, Aft => 3, Exp=> 1);
  New_Line;                                             -- 3.142E+0
  Length_IO.Put(2.78159, Fore=> 5, Aft => 3);
  New_Line                                              -- ¶¶¶¶2.750
End Main;
```

لاحظ أنّه في حال كان عرض النتيجة أكبر من العرض Width المحدد، فعندها يهمل العرض المحدد، ويتم إخراج النتيجة بالعرض اللازم لإخراجها. لاحظ أيضاً آخر عملية إخراج وفق Put، إذ أنّ القيمة المطبوعة ليست نفسها القيمة المحددة عند استدعاء التعليمة Put، والسبب في ذلك هو أنّ القيمة المحددة ليست من مضاعفات العدد 0.125 والذي يحدد النقطة الثابتة للنوع Length، إذ يصار دائماً إلى تقريب أي عدد ممثل بالنقطة الثابتة إلى أقرب قيمة له يمكن تمثيلها بالنقطة الثابتة. فالعدد 2.750 هو أقرب عدد لـ 2.78159 الممثل بالنقطة الثابتة من النوع Length المعرف.

وتحتوي Text_IO على توابع مشابهة من أجل الأنواع المرقمة. ويجب الحصول على نسخة مؤقتة من الحزمة البرمجية المولّدة Enumeration_IO من أجل تعيين نوع

المعطيات. وبشكل مشابهٍ للدخل/الخرج للنوع الرقمي، فالدخل من أجل الأنواع المرقمة يكون بشكلٍ حر، وذلك باستخدام البرنامج الجزئي Get . وعلى أي حال، فالتحميل الزائد للإجرائية Put يتضمن بعض معاملات التنظيم الإضافية. ومن أجل الدخل/الخرج للأنواع المرقمة، فالإجرائية Get لا تفرق بين الأحرف الكبيرة والأحرف الصغيرة. وبالتصريحات التالية يمكن استخدام أشكال الإجرائية Put :

```
with Text_IO;
procedure Main is
 type Status is (Normal, Warning, Alarm);
 package Status_IO is new
        Text_IO.Enumeration_IO(Enum=> Status);
 use Status_IO;
 use Text_IO;
Begin
-- Operation                          Display
    Put(Normal);                  -- NORMAL
     Put(Warning, Width => 10);       -- WARNING¶¶¶
Put(Alarm, Width => 7, Set => Lower_Case); -- alarm¶¶

  New_Line;
 End Main;
```

وإنّ أياً من الحزم البرمجية المولّدة الأربعة تزود بعمليات الإجرائيتين Get وPut، إذ يتم بواسطتهما قراءة وكتابة سلاسل محرفية، بدلاً من ملفات. ومن أجل البرنامج التالي:

```
with Text_IO;
procedure Main is
 type Mass is digits 10;
 package Mass_IO is new
          Text_IO.Float_IO(Mass);
 Buffer : String(1..10);
Begin
 Mass_IO.Put(To=> Buffer, Item => 3.14159,
                 Aft=> 3, Exp=> 1);
```

```
 Text_IO.Put_Line(Buffer);
End Main;
```

فإنّ النتيجة ستكون مايلي: ¶¶3.142E+0 .

وبما أنّه قد تمّ تحديد حجم المتغير Buffer فلا حاجة لتحديد عرض الحقل Fore، وبواسطة Aft وExp قد تم تحديد عدد الفراغات اللازم استخدامها.

وأخيراً، فإن المعاملات التالية لها قيم بدائية، والتي يمكن تغييرها من قبل المبرمج:

- Width وBase من أجل النوع الصحيح.
- Fore وAft وExp من أجل نوع النقطة الثابتة، ونوع النقطة العائمة.
- Width وSetting من أجل الأنواع المرقمة.

والقيم البدائية هذه ليست صريحة في توصيف مختلف البرامج الجزئية. بينما البرامج الجزئية تعود لمتغيرات تمّ تعريفها في حزم برمجية مختلفة، وهذه المتغيرات يمكن تغييرها. ومثال ذلك على الـ Enumeration_IO ما يلي:

```
Default_Width : Field :=0;
Default_Setting : Type_Set := Upper_Case;
procedure Put(Item  : in Fnum;
          Width : in Field  := Default_Width;
          Set   : in Type_Set := Default_Setting);
```

ويمكن تغيير القيم البدائية للمعاملات كما يلي:

```
with Text_IO;
procedure Main is
 type Status is (Normal, Warning, Alarm);
 package Status_IO is new
          Text_IO.Enumeration_IO(Enum=> Status);
Begin
 Status_IO.Default_Setting := Text_IO.Lower_Case;
 Status_IO.Put(Warning);
 Text_IO.New_Line;
End Main;
```

وإنّ استخدام المتغيرات لإعطاء وتغيير القيم البدائية للبرامج الجزئية يعتبر تقنيةً ناجعةً ونافعةً بشكلٍ جيد جداً، وهذا ما يعطي مرونةً أكثر من الترميز الصعب البدائي في البرامج الجزئية.



19

دورة حياة البرمجيات مع ADA
The Software Life Cycle

مرحلة التحليل

مرحلة تعريف دفتر الشروط

مرحلة التصميم

مرحلة الترميز

مرحلة الاختبار

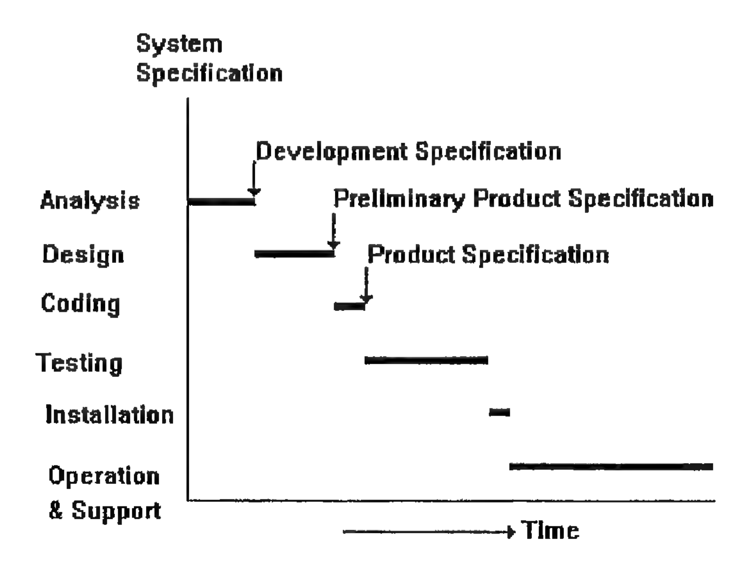
مرحلة التشغيل و الصيانة



تمتد دورة حياة البرمجيات منذ بدء تصميمها و حتى آخر استخدام لها. عادة، فإن مطوري البرمجيات يأخذون رؤية محدودة من دورة حياة البرمجيات، ويعالجون كل مرحلة من المراحل و كأنّها جزء مستقل. و هكذا، يمكن للمنفذين تصميم نظام باستخدام تقنية واحدة، الترميز بلغة أخرى، و من ثمّ يفحصون النظام بأكمله، مستخدمين لذلك مجموعة أدوات مختلفة كلياً. وتؤدي هذه الطريقة للعديد من المشاكل، إذ أنّ أقلها يتمثل بالتحكم بالتشكيلات المعقّدة، وبمجموعة وحدات برمجية لا تعمل مع بعضها البعض. وفي النهاية، يمكن للمطورين أن يُكملوا النظام، ولكن من المحتمل ألا يكونوا، في الوقت المناسب وبدون أسف، راضين عنه. ويمكن أن تقسم دورة حياة برنامج إلى ست مراحل أساسية. وحسب المؤلف، يمكن أن تتغير أسماء هذه المراحل بشكل طفيف، ولكن من أجل احتياجاتنا، فإننا ندعو هذه المراحل مايلي:

• التحليل • تعريف دفتر الشروط • التصميم • الترميز • الإختبار • التشغيل والصيانة.

والشكل التالي، يمثل تطور هذه المراحل حسب الزمن:



الشكل ١٩ ـ ١. يمثل سير هذه المراحل مع الزمن.

ولا يمكن للنظم الضخمة أن تقفز لوحدها فجأة. بل إنها تتطور من نظم صغيرة. والوضع المثالي لهكذا نظم، هو أن تتطور باستخدام أدوات و ترميزات متماسكة. ووفق هذا المعنى، يجب ألا نرمّز إلا بلغاتنا، و يجب أن نصمم باستخدام هذه اللغات أيضاً. ففي لغة ADA، نملك حالياً لغة يمكن تطبيقها بطريقة متماسكة، خلال دورة حياة البرنامج بكاملها. ومثلما ناقشنا وفي الفصل الثالث، تمثل ADA لغة تصميم؛ وفي هذا الفصل، سنفحص بالضبط، ما تعنيه هذه التعليمات.

١٩ - ١ - مرحلة التحليل (Analysis Phase):

خلال مرحلة التحليل لدورة حياة نظامنا، نبحث عن فهم أفضل لطبيعة المسألة، وتحديد المدى الذي نرغب فيه استخدام حل مؤتمت. وحالما تحدد حاجة الأتمتة، عندها نحجز وظائف خاصة للبرنامج، ويتم توثيقها عادة على شكل توصيف نظام. وخلال هذه المرحلة، نبدأ أيضاً بتعريف الموارد اللازمة لتكميل الحل، كقدرات (تحقيق) زرع البنية الصلبة و الوسائل البشرية في البرمجيات.

سيكون من غير المناسب التفكير باستخدام ADA في هذه النقطة من دورة الحياة. والسبب بسيط، فاللغة هي جزء من الحل، وليست جزءاً من فضاء المسألة. ويجب تنفيذ مرحلة التحليل بشكل مستقل عن أية لغة برمجة، وبالتالي، فإن استخدامنا للغة ADA في هذه النقطة، سابق لأوانه. وهذه التعليمات، يمكن أن تبدو متناقضة مع ما ذُكر في فقرتين سابقتين حول استخدام ADA خلال دورة الحياة. وعلى أي حال، إذا كنا نملك ASPE (ADA Programming Support Environment) صحيحة وصالحة، فسنطور توصيف النظام باستخدام وسيلة التوثيق من أجل إمكانية مقارنة حلّنا فيما بعد، مع دفتر الشروط الفنية.

١٩ - ٢ - مرحلة تعريف دفتر الشروط

(Requirements Definition Phase):

تبدأ عادة هذه المرحلة، بالموافقة على توصيفات النظام، وتنتهي بنجاح المراجعة النهائية للتصميم الأولي. والغاية من هذه المرحلة، هي تحديد دفتر الشروط الفنية المفصّل للبرنامج إنطلاقاً من الوظائف المحجوزة من خلال توصيفات النظام.

وفي الواقع، من الصعب تلافي تصميم حل كامل في هذه النقطة، ولكن يجب علينا أن نبقى بمستوى تصميم عال. والمنتج الأولي من هذه المرحلة، يتمثل بالتوصيف والموافقة عليه.

وفي هذه النقطة، سنفترض بأننا اخترنا لغة ADA كلغة تصميم. وبما أنّ المنتج من هذه المرحلة يمثل تصميماً أولياً، يمكننا استخدام توصيفات وحدات برامج ADA لتوثيق هذا المستوى العالي من نظامنا. وتماماً كما فعلنا في جميع مسائل التصميم السابقة، فإننا ننصح المصمم في هذه النقطة، بأن يأخذ رؤية تصريحية من الحل، و ليست الرؤية الأساسية التي تجبرنا بها لغات برمجة متعددة أخرى. وقد لاحظ Wheeler، بأنّ «القيود على بنية النظام المفروضة باستخدام لغة ADA كوسيلة لتوثيق تصميم النظام، لا تجعل فقط النظام أسهل تصميماً وتنفيذاً، بل تجعل النظام الناتج أكثر قابلية للصيانة».

وتضعنا هذه الخطوة على طريق تطوير متكامل لدورة الحياة، لأننا نملك الآن لغة تعبير متماسكة. وكما سبق، مع أدوات البيئة APSE المناسبة، يمكننا استخدام برمجية لإدارة التشكيلات، لمساعدتنا بالحفاظ على أثر كل تكرار وتطور التصميم. وبالإضافة لذلك، فإنه يمكننا الإستمرار باستخدام APSE لمتابعة دفتر الشروط الفنية للنظام، بالحجز الوظيفي لكل وحدة برمجية.

١٩ ـ ٣ ـ مرحلة التصميم (Design Phase):

تبدأ مرحلة التصميم للتطوير البرمجي، باكتمال دفتر الشروط، وتنتهي باكتمال نجاح تنقيح التصميم النهائي. ويتمثل المنتج الأساسي للتوثيق بمجموعة أولية لتوصيفات المنتج، والتي تعكس تصميم المنتج البرمجي. والهدف الأساسي من هذه المرحلة، يتمثل بتطوير تصميم مفصل لحلّنا البرمجي، متضمناً تعريف العلاقات (واجهات) بين الوحدات، ومخطط تدفق إجرائي مفصل.

وبما أنّ المستخدم لا يستطيع فهم أو إدراك جميع متطلبات وشروط البرنامج خلال المرحلة السابقة، فمرحلة التصميم عادة، تمثل معالجة تكرارية، متضمنة

تغيرات في توصيف التطوير، وتغيرات موافقة في التصميم. وهكذا تغيرات، إن لم تكن أساسية، تمثل إشارة جيدة بأنّ المطور و المستخدم قد حققا اتصالا صحيحاً. وأفضل وسيلة للإدارة يمكن استخدامها خلال هذه المرحلة، تتمثل بالمراجعة البنيوية لتصميم البرمجيات، والتي تحاول أن تجعل عملية التصميم أكثر رؤية، وأن تقود، كما نرغب، لمنتج أكثر فهماً.

وخلال هذه المرحلة، نبدأ باستخدام لغة ADA بشكل مباشر. في السابق، كنا نوثّق تصميماتنا البرمجية مع مخططات تدفق أو بـ PDL (Program Design Language) مختلفة. ومع ADA، يمثل التنفيذ النهائي توسيعاً للغة تصميم برنامجنا. وفي الواقع، توجد طريقتان يمكننا اعتمادهما مع ADA على شكل PDL. فالطريقة الأولى، حيث يتمثل أفضل توضيح بالخطوة المستخدمة في قسم النظم الفيدرالية في IBM، تستخدم مجموعة جزئية من التصميم في ADA كلغة تصميم، ومزودة بتوثيق مفصل. في الطريقة الثانية، فقد اقترح Hart، استخدام شكل أكثر حرية من ADA/PDL، الذي يسهّل بعض القواعد الدلالية.

وإن طريقة الـ IBM، تؤكد على التحليل الهرمي، وتعريف واجهات التخاطب، والوحدوية. وعلى أي حال، إن استخدامها يتطلب الإنتباه إلى قواعد و دلالة لغة ADA. ولذلك تتطلب تدريباً أكثر شمولية في لغة ADA، قبل أن تستخدم بنجاح. وإن طريقة Hart، لا تتطلب الإهتمام الكبير بلغة ADA نفسها، وبالتالي، يمكن استخدامها مع تدريب أقل. وإن كلا الطريقتين صالحتان، ولكن نفضل استخدام المجموعة الجزئية الصحيحة قواعدياً في لغة ADA، كلغة تصميم.

وعند الإعتناء التام بتسمية الأغراض في وحداتنا البرمجية، نجد بأنّ التصميم بلغة ADA، يصبح عملياً توثيقاً ذاتياً، حتى من أجل النظم ذات الحجم الوسط. وعندما ندخل إلى عالم الحلول الضخمة البرمجية، يصبح التوثيق الداخلي لقرارات التصميم أساسياً. ومهما كانت طريقة التصميم المستخدمة، فإننا نقترح دائماً استخدام رؤية تصريحية للحل، على الأقل في المستويات العالية، والتي من أجلها نستطيع استخدام

طريقة تصميم غرضية التوجه. ومن أجل المستويات المنخفضة، والمستويات الوظيفية المحضة، تعتبر تقنيات التحليل البسيط التي من النوع Top_Down، فعالة بنجاح.

ومثلما تمّ توضيحه في كلٍ من مسائل التصميم السابقة، يمكننا خلق تصميمنا البرمجي، باستخدام توصيفات وحدوية، من أجل تعريف واجهات التخاطب رسمياً بين الأغراض في مجال الحل. وفي هذه النقطة، يمكننا تقديم الأرومات (Stubs) لأجسام الوحدات، مع تعليقات مناسبة مشيرة للتصميم و المتطلبات للتنفيذ النهائي. والفائدة من هذه الطريقة تتمثل، حتى في هذه النقطة المبكرة، بأنّه يمكننا ترجمة النظام (باستخدام طريقة الـ IBM) و نترك اللغة تضبط المتناقضات المنطقية. فإذا كان لدينا أدوات الـ APSE المناسبة، فإنه يمكننا استخدام بعض برمجيات إدارة التشكيلات، للتحكم بنسخ الوحدات المنتهية. وبالإضافة لذلك، يمكننا الإستمرار بمتابعة المتطلبات، واستخدام الـ APSE، لتطوير بقية التوثيق الخارجي.

ويجب على المطورين، التحقق من أنّ تلك الطريقة مكلفة. وقد أظهرت التجربة، بأنّه عند استخدام لغة ADA كلغة تصميم، وفق تلك الطريقة، فإن كثيراً من الموارد تستهلك خلال مرحلة التصميم. وعلى أي حال، لقد أظهرت التجربة بأنّ هذه الطريقة تقدم فوائد ضخمة أكثر من الكلفة. وبشكل خاص، إن استخدام ADA كلغة تصميم، يمكن أن يحسّن نوعية التصميم، وذلك بإبراز واجهات التخاطب، ومكاملة طرح العديد من اقتراحات تصاميم هامة. وبالإضافة لذلك، إذا استخدمت لغة ADA بكاملها كلغة تصميم، يمكننا ترجمة تصميمنا في هذه النقطة، وكشف وتصحيح مشاكل واجهة التخاطب مبكراً، في دورة الحياة. وأخيراً، توجد وسائل توثيق قادرة على عبور الوحدات البرمجية، وتستخلص الأجزاء الهامة، بما في ذلك التعليقات. و هكذا، من الممكن توليد وثائق التصميم بطريقة شبه آلية، متلافين الكثير من الملل من عمل المطوّر.

١٩ ـ ٤ ـ مرحلة الترميز (Coding Phase):

تبدأ عادة مرحلة الترميز بعد الإكتمال الناجح لمرحلة التصميم و انتهائها... غير المعروف تماماً. وبمعنى آخر، لا تنتهي مرحلة الترميز حقيقة، حتى نترك قطعة

البرنامج الموافق للمنتج من التوثيق الأساسي من هذه المرحلة، ويتمثل بالتوصيف النهائي للمنتج، الذي يشير بأن المنتج البرمجي، قد تمّ بناؤه. والغاية من هذه المرحلة، يتمثل بتنفيذ التصميم المولّد بالمرحلة السابقة.

وباختيار ADA كلغة تنفيذ، فإنّ مرحلة الترميز تكون بسيطة نسبياً، وبالفعل تكون، امتداداً للمعالجة التي بدأت خلال التصميم. وقبل البدء بمرحلة التصميم، يجب أن يكون لدينا توصيف مفصل، على شكل مجموعة من وحدات برمجية بلغة ADA، مع توصيفات كاملة. وخلال مرحلة الترميز، نحتاج فقط لتكميل تنفيذ أجسام الوحدات البرمجية الخاصة بنا. وبالطبع، وبتنفيذ هذه الأجسام، ستظهر الحاجة لتحليل جديد. ومن وجهة النظر هذه، لا يوجد اختلاف بين المراحل التصميم/الترميز/الإختبار؛ وعلى العكس، ستشكل هذه المراحل معالجات تكرارية، في كل مستوى من الحل.

فإذا لم تعتمد ADA كلغة تنفيذ، كما هو الحال في عدم وجود لغة ADA على الآلة، فإنه يمكننا أخذ التصميم بلغة ADA، وتحويله إلى اللغة الهدف. وفعلياً، إن هذا ليس مختلفاً عن استخدام الـ PDL الخارجي، إذ يجب علينا دائماً إجراء بعض الشيء من التحويل. وعلى أي حال، لاحظ بأنّه من الأسهل الذهاب من لغة بنيوية كلغة ADA، إلى لغة غير بنيوية كلغة المجمع.

ويمكننا بسرعة، تطوير طريقة لتحويل تصميمنا من لغة ADA للغة أخرى عالية المستوى، مثل Fortran أو Pascal. ونعتقد أنّ عملية التحويل يمكن أن تؤتمت بحدود 75% (حسب التطبيق)، عند التغير إلى التنفيذ بـ Fortran أو Pascal. وكما هو الحال في المراحل السابقة، فإنه يمكننا استخدام الـ APSE الخاص بنا، لحفظ التحكم بالنسخ، ومساعدتنا بتحضير الوثائق. إن وسائل الـ APSE المتقدمة، مثل محررات القواعد الموجهة، يمكنها جعل مراحل التصميم/الترميز أكثر سهولة، عن طريق فقط، خلق وحدات برمجية صحيحة قواعدياً.

١٩ - ٥ - مرحلة الاختبار (Testing Phase):

إن الغاية من هـذه المرحلـة، تتمثـل بـالتحقق مـن تنفيـذ الـبرمجي وفقـاً لدفـتر الشروط الفنية، المقبول في توصيفات التطوير. ويجب ألا تنتظـر هـذه المرحلـة اكتمـال الترميز جميعه. فخطوة من النوع، وقليل من التصميم، وقليل مـن الترمـيز، وقليـل مـن الإختبـار، تعتـبر أكـثر تفوقـاً. فوفـق هـذه الطريقـة، نسـعى لتصحيـح أخطـاء الترمـيز والتصميم، في لحظة أكثر فائدة.

المطورون عادة، يقسمون مرحلة الإختبار لقسـمين، إختبـار الوحـدة، و اختبـار النظام. ونستطيع مباشرة إنجاز إختبار الوحدة، وفق طريقـة التصميـم/الترمـيز/الإختبـار المتزايدة. وخلال كل تكرار، نفحص بشكل تدريجي جزءاً أكثر فأكثر أهمية من النظـام المتكامل التام. وبعد الحصول علـى اختبـار الوحـدة، ننتقل إلـى اختبـار النظـام، مما يستوجب فحص النظام في بيئة تطابق أكثر ما يمكن، البيئة العمليـة. وفـي كـثير مـن الحالات، فإن هذا يتطلب تشغيل النظام أثناء الأزمة، على التوازي مع نظـام موجـود. وإن منتجات الوثائق الأساسية لهذه المرحلة، تتضمن تقارير الإختبار واعتماده.

وقد لاحظنا منذ قليـل، كيـف يمكـن لــ ADA و لــ APSE، أن تدعمـا مراحـل التصميم/الترميز الإختبار المـتزايدة. وخـلال مرحلـة الإختبـار، يمكننـا أيضـاً اسـتخدام أدوات APSE أخـرى، كـبرامج مسـاعدات التصحيـح الرمزيـة. وعـادة، تتمثـل إحـدى الصعوبات الأساسية خلال هذه المرحلـة، بحفـظ تشـكيلة رئيسـية للنظـام. وفـي نظـام ضخم، تتعقد المسألة، عندما تكشف الإختبارات ضـرورة إجـراء تغيـيرات صغـيرة فـي وحدات برمجية معزولة. ومرة ثانية، إن وسيلة تحكم بالتشكيلة مـن APSE، ستسـاعد بإدارة جميع النسخ المختلفة من البرمجي. وبالإضافة لذلك، وبما أنّ اللغة تدرك وجود مكتبة برنامج، فإنّ الـ APSE، ستشير للوحـدات التـي يجـب أن تعـاد ترجمتهـا، إذا حصل تغير في وحدة برمجية أخـرى. ويبـدو هـذا صعبـاً، ولكـن بالحقيقـة هـو سـهل نسبياً، لأنه نتيجة طبيعية لقواعد لغـة ADA، والتـي وفقهـا، تكـون الإرتباطـات بيـن الوحدات مسمّاة صراحة، في عبارات سياق الوحدات.

١٩ ـ ٦ ـ مرحلة التشغيل و الصيانة

(Operation & Maintenance Phase):

تبدأ هذه المرحلة بعد الحصول على النجاح النهائي لاختبارات النظام. وفي الواقع، إن هذه الحالة شبه مثالية، لأنّه، عملياً، غالباً ما يستخدم المطورون نظاماً بشكل عملي، قبل اكتماله أو اختباره كاملاً. فإذا استخدمنا تكراراً في التصميم /الترميز/ الإختبار، فإن هذه ليست مشكلة، ما دمنا نثق بالتوابع المختبرة حتى الآن. وعادة، مرحلة الصيانة هي الأكثر تكلفة بدلالة الموارد المستخدمة.وعادة، لا تنتهي مرحلة الصيانة هذه مطلقاً، ولكنها بالأحرى تخمد. وفي بعض الحالات، يمكننا التصريح بأنّ البنية الصلبة المتوفرة مهملة، مما يبدّد النظام بأكمله. ولكن في كثير من الحالات، نضع أصغر النظم في نظم كبيرة.

وبما أنّ الصيانة تحتاج لمعالجة التصميم/الترميز/الإختبار، فكل ما ذكرناه حتى الآن باعتماد لغة ADA، يمكن تطبيقه، وبشكل مخالف لبقية معالجات التصاميم، حيث يتم توثيق التصميم بشكل مختلف من لغة التنفيذ. وعندما نغير التنفيذ في ADA، نكون بالفعل قد غيرنا التوثيق. وبالإضافة لذلك، وبما أنّ لغة ADA لغة بنيوية عالية، فمن السهل الحفاظ على البنية الأصلية للنظام، من خلال إجراء تغيير قطع صغيرة منه. وكما في بقية المراحل، يمكن لـ APSE، أن تساعدنا بصيانة تحكم تشكيلات الوحدات البرمجية.

ومن الواضح، أننا قدمنا شرحاً مبسطاً لبعض النشاطات في دورة حياة البرمجي؛ إن وصفاً كاملاً" يتطلب عدة مجلدات. وكما رأينا، يمكن تطبيق ADA طيلة دورة حياة البرمجي. وبشكل مخالف لبقية النظم البرمجة، تقدم لنا ADA، لغة تعبير متماسكة، لتعطينا الإستقرار خلال عملية التطوير.



20

البرمجة على نطاق واسع
Programming in the Large

إدارة فضاء الأسماء

الترجمة المنفصلة

بنية النظم الضخمة



لقد اختبرنا حتى الآن، معظم بنى لغة ADA، وأوضحنا استخدام هذه البنى من خلال عدد من الأمثلة الصغيرة. وعلى أي حال، لقد تمّ تصميم لغة ADA، ليتم تطبيقها ليس فقط على المسائل الصغيرة، ولكن أيضاً، في مجالات يمكن أن تحتوي حلولها مئات الآلاف، وربما الملايين، من أسطر الترميز. ولا يمكننا ببساطة، توسيع تجربتنا لمسائل صغيرة، إلى تطبيقات ذات حلول ضخمة. ولحسن الحظ، مثلما سنرى في هذا الفصل، تقدم لغة ADA العديد من التوابع الوظيفية، لتساعدنا في التصدي لتعقيد النظم الضخمة.

٢٠ ـ ١ ـ إدارة فضاء الأسماء (Space Name Management):

في أي تطبيق هام، يجب أن يتكون الحل من عدة وحدات برمجية، بما في ذلك البرامج الجزئية، والحزم البرمجية، و المهام. فإذا تعاملنا مع نظم ضخمة، تحتوي عدة آلاف من أسطر الترميز، فيمكننا تجزئة الحل إلى مئات إن لم يكن إلى آلاف من الوحدات. وبالتالي، وبدون شك، سيكون هنالك عدد لا بأس به من المبرمجين، الذين يقومون بتطوير البرمجي. وفي نظم ضخمة كهذه، سيكون هنالك مئات من الأغراض البرمجية، توافق تجريدنا لأغراض العالم الحقيقي؛ وستكون الحالة ذاتها، من أجل البرامج الجزئية و المهام التي تشير إلى أعمال مجردة. وكنتيجة لذلك، ستحتوي برمجتنا عدة أسماء كيانات مصرح عنها.

فإذا كنا نستخدم لغة برمجة من الجيل الأول، مثل FORTRAN أو COBOL، فسيُجبرنا التركيب البنيوي لتلك اللغة، بجعل معظم تلك الأسماء عامة للنظام بأكمله. و بالتالي، في المشاريع التي تستخدم هكذا لغة، سنجد لوائح ضخمة من قواميس المعطيات، تصف استخدام كل اسم. وقبل استخدام أو التصريح عن كيان برمجي، يجب على المبرمج مراجعة اللوائح، ليتأكد من استخدامه الصحيح للاسم في السياق المناسب، أو من أنّه لم يحاول التصريح عن كيان باسم متعارض. وبما أنّ هذه الأسماء ستكون جميعها عامة للنظام بأكمله، فإن محاولة تحديد التأثير الناتج عن تغير كيان واحد، سيتطلب دراسة جميع مكونات النظام. وندعو المجال الذي تكون فيه جميع

الأسماء مرئية في نقطة واحدة، بمجال الأسماء. ومن البديهي، أن إدارة مجال الأسماء في نظم ضخمة في الـ Fortran أو في Cobol، ليست مهمة سهلة.

وإن استخدام لغة ADA في نظم برمجية ضخمة يستلزم أيضاً في حلوله، مئات من الكيانات المسماة. وعلى أي حال، تسمح لنا لغة ADA بمداخلة وتحزيم الوحدات البرمجية، وذلك بجعل الوظائف الضرورية فقط، هي المرئية في مستوى محدد، و تخفي التفاصيل غير الضرورية. ومثلما سنرى في المقطع التالي، تقدم ADA بعض قواعد مدى و رؤية بسيطة نسبياً، شبيهة بقواعد ALGOL لتساعد بإدارة مجال الأسماء. ولقد عرضنا منذ قليل معظم هذه القواعد بشكل غير رسمي؛ وستقدم هذه المقاطع بقية التفاصيل.

المدى والرؤية (Scope & Visibility):

بعبارات شكلية، إن مدى كيان هو منطقة من نص البرنامج، حيث يوجد أثر لتصريحه فيها. ومن جهة أخرى، تعرف رؤية كيان في أي مكان نستطيع الرجوع لإسمه. وفي جميع الحالات، يكون الكيان مرئياً من خلال مداه الطبيعي. وفي المقطع التالي، سنختبر كيانات أسماؤها مخفية أو محملة زائداً، وبالتالي، تكون رؤيتهما محدودة. والآن، على أي حال، دعنا نعالج حالات بسيطة.

إن مدى مُعرِّف (Identifier)، يبدأ بعد التصريح عنه مباشرة (التسمية للمرة الأولى)، و يمتد حتى آخر البرنامج الجزئي، أو الحزمة البرمجية، أو المهمة، أو الكتلة التي تحتوي التصريح. وأكثر من ذلك، فمن أجل المُعرِّفات المقدمة في توصيف برنامج جزئي، أو حزمة برمجية، أو مهمة، يمتد مداها إلى آخر جسم الوحدة الموافقة. ومع ذلك، فالعكس ليس صحيحاً. فإن مدى مُعرِّفاً تمّ تقديمه في جسم وحدة، يكون محدوداً بذلك الجسم و ليس مرئياً في التوصيف.

وقد تبدو الشكلية متناقضة، لذلك سنفحص بعض الأمثلة، والتي تُشير الأقواس فيها إلى مدى الكيانات ضمن نص البرنامج:

```
procedure Main is
 Object_1 : Integer;
 procedure Inner_Procedure is
  Object_2 : Integer;
 Begin
  -- Body of Inner_Procedure;
 End Inner_Procedure;
Begin -- Main
 Inner_Block:
 declare
  Object_3 : Integer;
 Begin
  -- Body Of Inner_Block
 End Inner_Block;
 -- Remainder of Main Body
End Main;
```

ففي هذا المثال، إن مدى Object_1 يمتد منذ تصريحه، و حتى آخر البرنامج الرئيسي Main. وأكثر من ذلك، إن Object_1 مرئي في مداه الطبيعي، لذلك، يمكننا الرجوع إليه بإسمه البسيط فقط. بينما ينحصر مدى و رؤية Object_2 داخل البرنامج الجزئي Inner_Procedure. ويمكننا الرجوع إلى Object_2 فقط داخل الإجرائية Inner_Procedure. وبشكل مشابه، فإن مدى و رؤية Object_3 تنحصر داخل الـ Inner_Block. وبما أن Object_1 يكون مرئياً في كل من Inner_Procedure وinner_Block، فيمكن بالتالي، استخدامه في جسم كل من الوحدتين.

وصورياً، إن Object_1 عام (ليس محلياً) بالنسبة لـ Inner_Procedure و لـ Inner_Block، بينما Object_2، يعتبر غرضاً محلياً بالنسبة لـ Inner_Procedure، أما Object_3، فيعتبر غرضاً محلياً بالنسبة لتعليمة الكتلة Inner_Block. وتسمح لغة ADA بالرجوع لأغراض غير محلية، وذلك ضمن مجال رؤيتها. وعلى أي حال، وفي

أي مستوى محدد، لا يمكننا الرجوع لكيانات مصرح عنها في وحدات متداخلة (لا يمكن الرجوع لكيان خارج مداه).

فإذا تمّ تقديم كيان في قسم تصريح برنامج جزئي، أو حزمة برمجية، أو مهمة، أو كتلة مسّماة، أو حلقة مسّماة، فيمكننا دائماً تسمية الكيان كمركب مختار، وذلك باستخدام إسم الوحدة كسابقة. وبالتالي، يمكننا الرجوع لـ Object_1 على طوال مداه Main.Object_1. وبشكل موافق، يمكننا استخدام الأسماء Inner_Procedure.Object_2 و Inner_Block.Object_3 داخل مدى الأغراض الموافقة. وكما سنرى في المقطع التالي، فإن هذه التسهيلات مفيدةِ جداً عندما نتعامل مع كيانات مخفية أو محملة زائداً. ونقترح استخدام هذه الرموز، في كل مرة تزيد فيها وضوحية الترميز.

ففي المثال السابق، كنا قادرين بالرجوع إلى كل كيان باستخدام إسمه البسيط؛ ولذلك نقول، بأنّ كل تلك الكيانات كانت مرئيةً مباشرة. وعلى أي حال، فإن مركبات التسجيلات، والحزم البرمجية، والمهام، ليست مرئية مباشرة، كما لاحظنا منذ قليل. وعلى سبيل المثال:

```
procedure Main
 type Node is (System_1, System_2, System_3);
 type Packet is
  record
   Source:     Node;
   Destination: Node;
   Message:   String(1..100);
  End record;
 My_Packet: Packet;
Begin
 -- Body Of Main
End Main;
```

وحسب القواعد المذكورة سابقاً، فلكلٍّ من الكيانات Node, Packet, My_Packet مدى ورؤية، يمتدان اعتبارا من نقطة التصريح الخاصة بكل كيان، وحتى نهاية البرنامج. وبالإضافة لذلك، فإن الرموز الثلاثة المرقمة System_1, System_2, System_3، لها نفس مدى Node. وبالمقابل، فإن مركبات Packet الثلاث المسماة Source, Destination, Message، ليست مرئية مباشرة. و بالتالي، فضمن Main، يمكننا الرجوع إلى هذه المركبات، كما يلي:

.My_Packet.Source, My_Packet.Destination, My_Packet.Message

وتطبق نفس القاعدة على المهام، والحزم البرمجية، كما هو موضح في المثال التالي:

```
procedure Main is
 Object_1 : Integer;
 package My_Package is
  Object_2 : Integer;
  procedure Inner_Procedure;
  End My_Package;
  package body My_Package is
  Object_3 : Integer;
  procedure Inner_Procedure is
  Begin -- Inner_Procedure
   -- remainder of Inner_Procedure
  End Inner_Procedure;
  -- remainder of My_Package body
  End My_Package;
  task My_Task is
   entry Path_1;
   entry Path_2;
  End My_Task;
  task body My_Task is
  Begin -- My_Task
   -- remainder of My_Task
  End My_Task;
Begin -- Main
 -- Body of Main
End Main;
```

وفي هذا المثال، إنّ Object_1 مرئـي فـي Main، ويمكـن الرجـوع إليـه داخـل الحزمة البرمجية My_Package، وداخل المهمة My_Task. وأكثر من ذلك، إن مـدى Object_2 يمتد داخل جسم الحزمـة البرمجيـة My_Package، بينمـا Object_3 ليـس مرئياً إلا في جسم الحزمة البرمجية My_Package. وبما أنّه قد تمّ التصريح عن كلٍّ من Object_2 و Inner_Procedure في توصيف الحزمـة البرمجيـة My_Package، فيمكـن استخدامهما داخل مـدى جسـم الحزمـة البرمجيـة My_Package. وبمـا أن، مركبـات التسجيلة، ومركبات الحزمة البرمجية (و مداخل المهمة) لا يمكن رؤيتهمـا مباشـرة، فبالتالي، يجب استخدام الترميز المركب المختار من أجل الرجوع لهذه الكيانات، من خارج الحزمة البرمجية (أو المهمة) نفسها. وهكذا، فإنه فـي Main، يمكـن الرجـوع للغـــرض My_Pacakge.Object_2 وللإجرائيـــــة My_Package.Inner_Procedure. وبالإضافـة لذلـك، يمكننـا الرجـــوع للمدخـــل My_Task.Path1 و للمدخـــل My_Task.Path2. وبما أنّـه وبشـكل دائم، تسـتخدم الحـزم البرمجيـة فـي نظـم لغـة ADA، فأسماء مركبات الحزم البرمجية، يمكن أن تصبح طويلة بشكل كافٍ، خصوصاً إذا كان هنالك تصريحات حزم برمجية متداخلة. وتتمثل الطريقة لتجنب هـذه الأسـماء الطويلة، باستخدام عبارة use. وفي أي منطقة تصريح، تكون فيهـا الحزمـة البرمجيـة مرئية My_Package، أو متبوعة بعبارة with مسمّاة بـ My_Package، يمكننـا إعطـاء التعليمة كما يلي: ;use My_Package.

واعتباراً من تلك النقطة، لسنا بحاجة لسبق المركبات بإسم الحزمـة البرمجيـة، إلا إذا كان هنالك إلتباس.

ويجـب عـادة، تطبيـق عبـارة use بحـرص. والرمـز "_" الفـاصل بيـن الأسـماء القصيرة، الذي يتوسط ترميز العامل، ويبدو قادراً على التبليغ في مكـان تصريـح كيـان التأثير على قابلية القراءة. وعبارة use تؤدي لالتباس مع زيادة عدد الكيانات في مجال الأسماء، ولتأثيرات جانبية عند تغير الترميز. فإذا صدرت حزمتان نوعاً يسـمى Mass، و تمّ تطبيق عبارة use على كلٍّ من الحزمتيـن، فـلا يمكـن رؤيـة كليهمـا، وذلـك لأنّـه يوجد هنا إلتباس في Mass، كما في المثال التالي:

```
package Measures is
 type Mass is digit 6;
...
End Measures;
package Weights is
 type Mass is (Grams, Kilograms, Pounds);
...
End Weights;
with Measures, Weights;
procedure Mix in is
 use Measures, Weights;
 Something: Mass;
Begin -- Mix_In
...
End Mix_In;
```

إن هنا التصريح عن Something غير صال،ح بينما Mass لا تصبح مرئية بشكل مباشر. وإن وجود نوعين في نفس المستوى و تحت نفس الإسم، ولكن بتمثيلين مختلفين، غالباً ما يشير إلى مشكلة في التصميم. فإذا تمّ تحقيق أي مرجع إلى أسماء يوجد فيها التباس، كما هو الحال في Mass في هذا المثال، فإنّ عبارة use تكتشف ذلك و تحدده في زمن الترجمة.

لاحظ عدم وجود بنية مكافئة من أجل المهام. ولذلك، يجب علينا دائماً الرجوع لمدخل مهمة باستخدام الترميز المركب المختار.

التحميل الزائد، والإخفاء، وإعادة التسمية

(Overloading, Hiding, & Renaming):

في أي نظام، نتجنب عدم توافق الأسماء. وفي الواقع، من أجل قابلية القراءة، فإننا نرغب بإعطاء لكل كيان إسماً يوضح غايته أو استخدامه. ولكن، وخاصة في النظم الضخمة، يمكن تطبيق نفس الإسم على كيانين مختلفين منطقياً. وكما سنرى، تقدم ADA بعض البنى لتتعامل مع هكذا حالات.

وعندما يستخدم نفس الإسم أو رمز العامل من أجل كيانات مختلفة مدى كل واحد متراكب مع مدى الآخر، عندها يقال، بأنّ الإسم محمل زائداً. فعلى سبيل المثال:

```
declare
 type Color is (Red, Green, Blue);
 type Light is (Red, Yellow, Green);
 Pixel: Color;
 Stoplight: Light;
Begin
...
End;
```

ففي هذه الحالة، إن الأسماء Red, Green تم تحميلهما زائداً، لأنه تمّ استخدامهما بسياقين مختلفين. وتسمح لنا ADA باستخدام هذه الأسماء بشكل حر، طالما لا يوجد التباس على Red, Green المذكورتين. وبالتالي، يمكننا كتابة ما يلي:

```
Pixel := Green;

Stoplight := Green;
```

ففي كلا الحالتين، يستطيع ضمنياً، مترجم لغة ADA تحديد أي Green يجب الرجوع إليها، وذلك بفحص نوع الطرف الأيسر من تعليمة الإسناد. فإذا لم نستطع استخدام السياق لإزالة الإلتباس، يمكننا الإشارة صراحة للنوع مع التعبير المؤهل، كما يلي:

```
Pixel := Color ( Red );
```

ويمكننا إجراء تحميل زائد على غير الأنواع المرقمة، مثل الكتل، والبرامج الجزئية، ومداخل المهام. فعلى سبيل المثال، يمكننا تعريف عدة برامج جزئية محملة زائداً تدعى Put، تعمل كل واحدة منها مع معاملات مختلفة:

```
Integer); : procedure Put(Element
```

```
procedure Put (Element: Float);

procedure Put (Element: Color);

procedure Put (Element: Light);
```

ومن المبرر هنا إستخدام أسماء محملة زائداً من أجل هذه الغاية. وبما أنّ الغرض أو العملية هي نفسها منطقياً، فمن الحكمة استخدام نفس الإسم. وعلى أي حال، لاحظ بأنّه يجب استخدام التحميل الزائد بحرص، لأنّه يؤدي لتشويش القارئ.

وفي المثال السابق، يمكننا وبحرية، استدعاء كل إجرائية، تاركين لقواعد اللغة تحديد أي برنامج جزئي نرجع له:

```
Put (367);          -- Put an Integer
Put (4.6);          -- Put a Float
Put (Blue);         -- Put a Color Value
Put (Light'(Green)); -- Put an Integer
```

لاحظ أنّه في المثال الأخير، قد استخدمنا تعبير مؤهل، لأنّ Put(Green) كانت ستؤدي لإلتباس. ويمكننا استخدام مؤهلات أخرى، مثل ترميز المركبات المختارة، أو استخدام إسم وحدة كسابقة، من أجل إلغاء الإلتباس بين الكيانات المحملة زائداً.

ويكون الإسم محملاً زائداً، إذا استخدِم لتمثيل كيانات مختلفة. وعلى أي حال، توجد بعض الصفوف من كيانات برنامج، مثل الأغراض، لا يمكن إجراء تحميل زائد عليها. فإذا استخدمنا اسماً في تصريح في بنية متداخلة، فإننا نخفي بالفعل كيان خارجياً بنفس الإسم. فعلى سبيل المثال، لنعتبر الترميز التالي:

```
procedure Main is
 My_Object: Boolean;
Begin -- Main
 Inner_Block;
  declare
   My_Block: Boolean;
  Begin -- Inner_Block
```

```
  -- body of Inner_Block
 End Inner_Block;
 -- remainder of Main body
End Main;
```

في هذا المثال، إن مدى Main.Object يمتد من نقطة تصريحه و حتى نهاية Main، بينما مدى Inner_Bloc.My_Object محدود بالكتلة نفسها. وقد تمّ استخدام الاسم My_Object من أجل كيانين مختلفين، وكذلك، إذا استخدمنا الإسم البسيط My_Object داخل الكتلة Inner_Block، فإنّ قواعد الرؤية في ADA تبين بأننا رجعنا للغرض المحلي (وهو Inner_Block.My_Object). وإن Main.My_Object بالفعل قد تمّ إخفاؤه. وإذا أردنا الرجوع إلى هذا الكيان غير المحلي، فإنه يجب علينا صراحة، استخدام ترميز المركب: Main.My_Object.

وطبيعياً، إن إخفاء كيانات برنامج يطرح مشكلة، لأن الرجوع إلى كيانات عامة غير مألوف كثيراً. فإذا أردنا الرجوع إلى غرض ما، فإنه يكون من المفضل استيراد الغرض من خلال، ربما، معامل برنامج جزئي. وبشكل عام، فكلما حللنا الحلول إلى مستويات أعمق فأعمق، فإنه علينا ألا نقلق بالأسماء التي نستخدمها في السياق الخارجي.

وعندما نبدأ التداخل في المستويات الأكثر عمقاً، على أي حال، فإننا نجد بأنّه إذا رجعنا إلى كيانات في سياق متداخل، فإن الأسماء ستصبح طويلة نسبياً، بسبب ترميز المركب المختار. ولتجنب هذه الحالة، تسمح لغة ADA بإعادة تسمية الحزم البرمجية. وعلى سبيل المثال، لنعتبر الحزمة البرمجية المتداخلة التالية:

```
package Math_Library is
 type Radians is..
 package Trig_Functions is
  function Cos (Angle : Radians) return Float;
```

```
 End Trig_Functions;
 package Matrix is
...
 End Matrix;
End Math_Library;
```

وفي حال غياب عبارة use، والتي تزيد حجم مجال الأسماء، يجب أن نرجع للوظيفة Cos كما يلي:

```
Math_Library.Trig_Functions.Cos(Some_Angle)
```

وعلى أي حال، يمكننا إعادة تسمية هذه الحزمة الداخلية، كما يلي:

```
package Trig renames Math_Library.Trig_Functions;
```

وبالتالي، يمكننا الرجوع للوظيفة Cos كما يلي:

```
Trig.Cos(Some_Angle)
```

ومثلما ناقشنا في فصل سابق، تعتبر إعادة التسمية من التقنيات الفعّالة للسيطرة على رؤية الكيانات، و خاصة الكيانات المصدّرة عبر حدود الحزم البرمجية. وتسمح لغة ADA بإعادة تسمية الأغراض، والإستثناءات، ومداخل المهام، و البرامج الجزئية. وبالتالي، يمكننا حل تضارب الأسماء، والحصول على تقييم جزئي، و تقديم مرادفات للكيانات. وعلى سبيل المثال، لتكن التصريحات التالية:

```
My_Array : array(1..10) of Integer;

Alarm : exception;

procedure Quick_Sort(Elements : in out List);
```

ويمكننا إعادة تسمية هذه الكيانات، كما يلي:

```
Your_Integer : Integer renames My_Array(3);

Water_Level : exception renames Alarm;

procedure Sort(Things : in out List) renames Quick_Sort;

procedure Quick_Sort(Elements : in out List) is separate;
```

ولقد عرّفنا إسماً جديداً للكيان المُصّرح عنه سابقاً. ففي الحالة الأولى، استخدمنا التصريح عن إعادة التسمية، لتقييم دليل المصفوفة مرة واحدة فقط. ولاحظ في مثال إعادة تسمية البرنامج الجزئي، أننا نستطيع استخدام أسماء مختلفة، من أجل المعاملات الصورية. ولاحظ بأنّه يجب أن يأتي جسم الإجرائية Quick_Sort، بعد إعادة تسميتها. التصريحات عن إعادة التسمية، تشكل جزءاً من التصريحات الأساسية، التي يجب أن تأتي قبل أي جسم.

بالإضافة لذلك، فإن تطبيق عدة قيم بدائية مختلفة، إن وجدت، يمكن إجراؤه على هكذا معاملات. ومثال ذلك ما يلي:

```
function Append (E1, E2 : Element) return List;

function "&" (Left, Right : Element) return
List renames Append;
function Center (The_Item :    String;

        Within_Length : Positive;

      dding_With : Character) return String;

function Pad (The_String : String;

         Location :    Positive;

         Pad_Character : Character := ' ')
    return String renames Center;
function Make_Title (The_Item : String;

         Length :    Positive := 80;

         Pad :       Character := '*')
return String renames Center;
```

والتصريحات عن إعادة التسمية، يمكن استخدامها أيضاً، لإعطاء أسماء بسيطة لمركبات تسجيلة، ومداخل مهام. وبالتالي، فإن هذا ما يؤدي غالباً إلى قلة فهم للترميز، ولكن يمكن أن يستخدم بشكل فعال على مدى صغير:

```
Set_My_Packet:
declare
 Source:  Node renames My_Packet.Source;
 Message: String renames My_PAcket.Message;
Begin
 Message(1..2) := "Hi";
End Set_My_Packet;
Ping_System_1:
declare
 procedure Ping renames My_Task.Path_1;
Begin
 Ping;
End Ping_System_1;
```

لاحظ بأنّ إعادة التسمية لـ Message، لا تتطلب قيداً على المجال، بينما إعادة التسمية لا تصرّح عن غرض جديد. في الواقع، لقد خلقنا إسماً جديداً لغرض موجود، وإن الغرض الموجود سابقاً له قيد على المجال. ويمكن أيضاً إعادة تسمية مداخل المهام كإجرائيات.

وإذا كنا بحاجة لتعريف إسم مختلف لنوع، فيمكننا الحصول على نفس الأثر، مثل التصريح عن إعادة التسمية باستخدام نوع جزئي، كما هو موضح في المثال التالي:

```
package Abstract_Type is
 type My_Type is...
End Abstract_Type;
subtype Local_Type is Abstarct_Type.My_Type;
```

وكلما استمرينا بخلق وحدات برمجية متداخلة، كما هو الحال في النظم الضخمة، فإننا نقوم بالفعل، بزيادة عدد العناصر في مجال الأسماء. وتسمح إعادة تسمية الأسماء باستخدام أسماء أقصر، ولكن لا تساعدنا دائماً بتحديد كمية الأسماء المرئية من نقطة محددة. ومن أجل مساعدة السيطرة على كمية الأسماء التي يجب التعامل معها من أجل مستوى محدد،

وتقليص موضع هذه الحالة، يجب أن نهتم بالترجمة المنفصلة لوحدات البرنامج.

٢٠ ـ ٢ ـ قضايا الترجمة المنفصلة

(Separate Compilation Issues):

في أي نظام برمجي مؤلف من العديد من الوحدات البرمجية، فإن جعل الوحدات مستقلة عن بعضها البعض، لا يعتبر فقط عادة جيدة، لكنه غالباً يكون ضرورياً: إذ يمكن لمجموعات مختلفة، أن تعمل على أجزاء منفصلة من الحل، وبالتالي، يمكن أن تكون كل وحدة برمجية في مراحل تطوير مختلفة. يتم تحسين الإختبارات، إذا كانت الوحدات البرمجية منفصلة فيزيائياً و مدروسة في عزلة تامة. وأكثر من ذلك، ففي نظام ضخم، سيكون غير فعال إعادة ترجمة البرمجي بأكمله، بسبب وجود تغييرات طفيفة في وحدات ذات مستوى منخفض.

فما هو مرغوب، عندها، يتمثل بإمكانية ترجمة الوحدات المختلفة من البرنامج بشكل منفصل. وبالطبع، لقد قدمت بقية اللغات هذه الإمكانية، ولكن، كما سنرى في هذا المقطع، تقدم لغة ADA بعض الوسائل الإضافية، لإدارة تعقيد الوحدات المستقلة.

الوحدات المكتبية (Library Units):

مثلما لاحظنا في فصل سابق، فإننا لسنا مجبرين في لغة ADA بترجمة برنامج كامل كل مرة، بالرغم من إمكانية تحقيق ذلك، إذا أردنا. و بالتالي، تسمح لغة ADA بإخضاع نص برنامج لترجمة أو أكثر. وتتألف الترجمة المحددة، من وحدة أو عدة وحدات ترجمة متضمنة:

- التصريح المولّد (Generic declaration).
- النسخ المؤقتة المولّدة (Generic instantiation).
- التصريح عن برنامج جزئي (Subprogram declaration).
- جسم برنامج جزئي (Subprogram body).
- التصريح عن حزمة برمجية (Package declaration).
- جسم حزمة برمجية (Package body).
- الوحدة الجزئية (Subunit).

لاحظ بأنّ المهمة لا تمثل وحدة برمجية. ومن أجل ذلك، من الشائع تضمين مهمة داخل حزمة برمجية، بطريقة نستطيع بها الحصول على وحدة مكتبية.

ويقال بأنّ الوحدات المترجمة لبرنامج، تنتمي لمكتبة برنامج. ومن وجهة نظر عامة، فإن كل وحدة ترجمة، تُدعى بوحدة مكتبية أو وحدة ثانوية (أجسام وحدات مكتبية، ووحدات جزئية). ويجب إعطاء اسم وحيد لكل وحدة. ومثلما ذكرنا سابقاً، إن الوحدة الرئيسية من أي برنامج، يجب أن تكون برنامجاً جزئياً، والذي بالتعريف، سيكون وحدة مكتبية لبرنامج. وإن قواعد اللغة، لا تعّين كيف يمكننا تعريف وحدة رئيسية؛ إذ أنّ اختياره يُترك للبيئة. وفي كل مرة يخضع بها نص برنامج للترجمة، تعالج ADA هذه الوحدات المترجمة وكأنّه مصرّح عنها في سياق الحزمة البرمجية Standard. وبالنتيجة، وباستثناء الأسماء المخفيّة، يمكننا استخدام كل الكيانات المعرّفة في Standard.

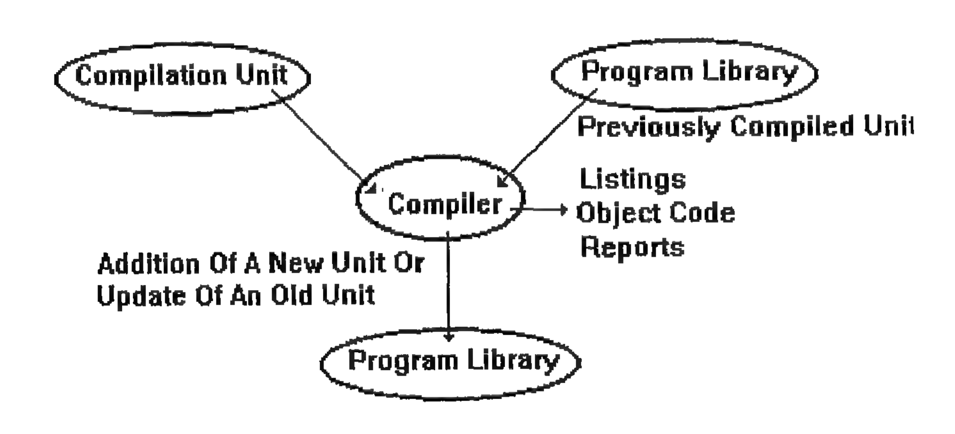
فإذا وُجدت بعض الوحدات المكتبية المترجمة سابقاً، يمكن لبقية الوحدات تطبيق عبارة with، للحصول على رؤية أي وحدة معطية. فإذا عدنا لمثالنا الأول في التصميم، يمكننا وبشكل مستقل، إخضاع الوحدات Words, Lines, Document, Concordance للترجمة. ويمكن لبقية الوحدات، الرجوع لهذه الوحدات بتوصيف السياق:

```
with Words, Lines;
package Concordance is...
```

واعتباراً من هنا، يمكن استخدام ترميز المركب المختار، وتصريحات إعادة التسمية، من أجل الحصول على رؤية مكونات الحزم البرمجية، أو يمكن تطبيق عبارة use، أو تصريحات إعادة التسمية، للوصول إلى الرؤية المباشرة. وفي أية حالة، تتمثل فائدة هذه الطريقة، بأنّه يمكننا وبشكل اختياري، جعل الوحدات المكتبية التي نريد استخدامها مرئية. ويمكننا استخدام عبارة with من أجل أية وحدة مكتبية، بما في ذلك الوحدات المولدة. والرجوع إلى وحدة مكتبية في عبارة سياق تعرف ارتباطاً بين الوحدات البرمجية، يؤثر على ترتيب الترجمة و إعادة الترجمة.

ولا يعتبر مفهوم الترجمة المنفصلة جديداً، ولكن، بعكس بعض لغات البرمجة الأخرى، تتطلب لغة ADA إجبار المترجمات بتحقيق قواعد الأنواع و كأنّ الوحدة غير منفصلة عن أصلها. ووفق هذه الطريقة، يتم حفظ الحماية التي يقدمها التنويع القوي، حتى إذا خضعت البرامج لعدة ترجمات منفصلة. وبالرغم من أن هذا يزيد تعقيدات مترجمات ADA، فإن هذه القاعدة مفيدة جداً من أجل بناء نظم ضخمة.

ومثلما هو مبين في الشكل ٢٠ ـ ١، فإن المترجمات تعرّف بوجود مكتبة برنامج. وعندما نُجري ترجمة، يمكن لعدة وحدات مختلفة تعيين استخدام الوحدة المكتبية في توصيف سياقها. وبعد الترجمة، نقدم سرداً، وتقارير، و من الممكن ترميز أغراض. بالإضافة لذلك، يمكن إضافة الوحدة المترجمة إلى المكتبة إذا كانت وحيدة، أو يمكن تغير وحدة قديمة. وفي أية حالة، إذا نجحت الترجمة، تصبح الوحدات المترجمة الخاضعة جزءاً من المكتبة، وتصبح صالحة للاستخدام من قبل وحدات أخرى.



الشكل ٢٠ ـ ١ نموذج الترجمة في ADA.

الوحدات الجزئية (Subunits):

يتم بناء البرامج باستخدام وحدات مكتبية، اعتباراً من وحدات منخفضة المستوى بفضل وحدات المكتبة. ومع الوحدات الجزئية، يمكن تطوير البرامج بطريقة هرمية، من الأعلى للأدنى. وعند تحليل نظام، يمكننا تصميم توصيف برنامج جزئي،

أو حزمة برمجية، أو مهمة، لكن، نستطيع تأخير تنفيذ جسم الوحدة. في لغة ADA، يمكننا خلق أرومات جسم يمكن ترجمتها منفصلاً، وندعو هذه الأجسام المترجمة منفصلاً، بالوحدات الجزئية.

فلنعتبر البرنامج الرئيسي التالي، والذي من خلاله، قمنا بتطبيق التحليل الوظيفي:

```
procedure Main is
 task Black_Box is
  entry Recive (Message : out String);
  entry Send  (Message : in String);
 End Black_Box;
 task body Black_Box is separate;
 package Transform is
  procedure Decypher (Message : in out String);
  procedure Encod   (Message : in out String);
 End Transform;
 package Transform is separate;
 procedure Report (Message : in String) is separate;
Begin -- Main
...
End Main;
```

ويمكننا إجراء ترجمةٍ منفصلةٍ لأجسام المهمة، والحزمة البرمجية، و البرنامج الجزئي، باستخدام الشكل التالي:

```
separate (Main)
task body Black_Box is
Begin
...
 accept Recive (Message : out String);
```

```
...
 accept Send  (Message : in String);
...
End Black_Box;
separate (Main)
 procedure Decypher (Message : in out String is
 Begin
. ..
 End Decypher;
 procedure Encode (Message : in out String is
 Begin
. ..
 End Encode;
End Transform;
separate (Main)
procedure Report (Message : in String) is
Begin
...
End Report;
```

وكل واحد من الأجسام الثلاثة هذه، يمكن ترجمته بشكل منفصل؛ وإن جميع الوحـدات الجزئيـة مـن Main، لا تحتـاج للترجمـة مـع بعضهـا البعـض. ووفـق هـذه الطريقة، نقدم أولاً ترميزاً أصغرياً أو معدوماً، لكل وحدة برمجية، التي نبدلها فيما بعد بتنفيذ كامل. لاحظ بأنّه يجـب ضـم عبـارة separate لتعريـف أصـل الوحـدة. وبشـكل مشابه لعبارة with، تعرّف عبارة separate ارتباطاً بيـن وحـدات البرنـامج. وبالإضافـة لذلك، يمكن أن تملك وحدة جزئية توصيف سياقها الخاص، الـذي يعـرّف الوحـدات المكتبيـة، والـذي يحتـاج لعبـارة with. وإن الأسـماء المنضـدة فـي عبـارة with هـذه،

ستكون مرئية للوحدة الجزئية، مع بقية الأسماء التي كانت مرئية في تلك النقطـة مـن أرومة الجسم في الأصل.

فإذا احتوت الوحدة الجزئية على وحدة جزئية وهكذا، فإنـه يجب أن تتضمـن عبـارة separate التوصيف الكامل لإسم الملف. وعلى سبيل المثال، إذا احتـوت Transform على وحدة جزئية، فإن عبارتها تكون: separate (Main.Transform)

ونتيجة لقواعد مكتبات ADA، هي أنّ جميع الوحدات الجزئية، و التي تملك نفس الوحدة المكتبية السلف، يجب أن تملك أسماءاً وحيدة. ويمكن أن تكون هذه مشكلة على النظم الضخمة، إذا لم تُدر بشكل تام.

ترتيب الترجمة، وإعادة الترجمة

(Order of Compilation & Recompilation):

عند بناء نظام من عدة مترجمات، تتطلب الإرتباطات المعرّفة صراحة بين الوحدات، بأن تترجم هذه الوحدات وفق ترتيب معين. وبشكل أساسي، حسب القاعدة، يجب أن تترجم كل وحدة قبل أن تكون مرئية بالنسبة لوحدة أخرى. وبشكل خاص، فإن توصيف برنامج جزئي، أو حزمة برمجية، أو مهمة، يجب أن تتمّ ترجمتها قبل الجسم الموافق لكلٍّ منها. بالإضافة لذلك، يجب أن تترجم الوحدة الأصل، قبل ترجمة وحداتها الجزئية. وأكثر من ذلك، فإذا سمّت وحدات ترجمة محددة، وحدات مكتبية أُخرى في عبارة with، فيجب ترجمة توصيفات تلك الوحدات المكتبية أولاً. ووفق قواعد رؤية ADA، فإن هذا يعني بأنّه يمكن ترجمة أجسام الحزم البرمجية، وفق أي ترتيب. وفي بيئة برمجية جيدة، تُقدّم الوسائل لمساعدتنا بالإلتزام بهذا الترتيب، بالرغم من أن مترجم ADA، يحذرنا من محاولة ترجمة وحدة، قبل ترجمة كل الوحدات المرتبطة بها.

وخلال تطوير نظام، سنُخضع بالطبع عدة وحدات للترجمة من جديد، مثلاً، لتصحيح خطأ، أو لتغير أو لإكمال تنفيذ. وتقدم تسهيلات الترجمة المنفصلة بلغة ADA أدوات فعالة لبنّاء النظام: فالتغير يكون محدوداً بوحدة برمجية واحدة. وبسبب

قواعد الرؤية، يمكن إعادة ترجمة جسم برنامج جزئي، أو حزمة برمجية، أو مهمة، أو وحدة جزئية، دون التأثير على أي وحدة أخرى (إلا إذا كانت أصل وحدة جزئيـة). و بالتالي، طالما لم نغير في توصيف وحدة برنامج، يمكننا وبحرية تغيير التنفيذ الموافق. ووفق هذه الطريقة، نحفظ بنية التصميم المنطقية للنظـام. فـإذا أجرينـا تغيـير لأسـباب الفعالية أو لتصحيح خطأ، تسمح لنا لغة ADA بتحديد موضع تـأثيرات التغيـير. فـإذا غيّرنا أصل وحدة جزئية، فإنه يجب إعادة ترجمة الوحـدة الجزئيـة؛ وفـي الحقيقـة، يجب إعادة جميـع الوحـدات الجزئيـة التابعـة للأصـل. وبالإضافـة لذلـك، إذا غيرنـا توصيف وحدة مكتبية، يجب أن نعيد ترجمة الجسم الموافـق، وبالتـالي، فـإن جميـع الوحدات التي تذكر هذه الوحدة المكتبية في عبارة with.

وإن الوحدات المولّدة و البرامج الجزئية المدرجة، تغير بدقة، هذه القواعد. وفي حالة العملي Inline، عرّفنا ارتباطاً بين توصيف وحدة برمجية، وجسمها. وعلـى وجـه التخصيص، فإنه يجب ترجمة الجسمقبل أن يرجع زبون إلى الوحدة. وبشكل مشابه، تتطلب بعض التنفيذات بأن تتم ترجمة التوصيفات المولّدة وأجسامها، قبل إجراء نسخ مؤقتة عنها.

والجدول التالي يلخص قواعد الترجمة في لغة ADA:

جدول إعادة الترجمة

إذا تمّ تغيير:	يجب إعادة ترجمة:
ـ التوصيف	ـ التوصيـف، وجسـم التوصيـف، وبشـكل تكـراري، جميع الوحدات التي ترتبط على هذه الوحدة.
ـ الجسم	ـ الجسم و وحداته الجزئية.
ـ الوحدة الجزئية	الوحدة الجزئية و وحداتها الجزئية.
ـ الجسم المولد	ـ الجسم و جميع نسخه المؤقتة.
ـ الوحدة المدمجة	ـ الجسم و جميع تكرارات الاستخدام

وبالرغم من أن تسهيلات الترجمة المنفصلة بلغة ADA تساعدنا على تحليل نظام ضخم إلى أجزاء يمكن إدارتها، فإنه يمكن أن تصبح إعادة الترجمة، عقدة من أجل تطوير النظم. وعلى سبيل المثال، يمكن أن يؤدي تغيير وحدة واحدة فقط، لإعادة ترجمة مئات من بقية الوحدات، مكلفة بذلك ساعات، و ربما أيام من زمن الترجمة. وبالتالي، عندما يتضخم حجم النظام، من الضروري أن يكون تصميم جميع الوحدات البرمجية أكثر ما يمكن بسيطاً لتحاشي إعادة الترجمات الهائلة.

وكما ناقشنا في فصول سابقة، تعتبر الدرجة التي من أجلها يجبر تغيير في وحدة برمجية، إعادة ترجمة وحدات أخرى، مقياساً جيداً لارتباط تصميم: إن مترجمات ADA و بيئات تطوير ADA، تقدم وسائل لتقييم الإرتباط، وبعد ذلك، يجب أن تجبر هذه الأدوات ترتيباً صارماً لإعادة الترجمة.

٢٠ ـ ٣ ـ بنية النظم الضخمة (Large Systems Architecture):

مثلما ناقشنا في فصل سابق، لا نستطيع أن نأمل إتمام نظام برمجي ضخم بنجاح، باستخدام طريقة تصميم غير نظامية. وأكثر من ذلك، لا يمكن مطلقاً بناء النظم الضخمة خلال مرة واحدة؛ إذ أنها تتطور بدءاً من نظم أصغر. المثالي، هو أن توازي بنية الحل، رؤية تطويرنا للعالم الحقيقي. وباستخدام المجموعة الغنية بوحدات برامج ADA، وتسهيلات الترجمة المنفصلة، يمكننا بناء حل بحيث توافق بنيته الأغراض والعمليات في مجال المسألة. وكما سنرى، تسمح لنا ADA بالتطوير البرمجي إعتباراً من الأعلى للأدنى (Top Down)، أو من الأدنى للأعلى (Bottom Up).

التطوير من الأعلى للأدنى (Top Down Development):

تتضمن طريقة التصميم "من الأعلى للأدنى" البدء بأعلى مستويات التجريد، ومن ثم تحليل النظام إلى مستويات أكثر بدائية، مثلما ناقشنا في فصل سابق، حيث أن طريقة تصميم Yourdon الوظيفي تتبع هذا النموذج.

وفي ADA، يمكن استخدام الوحدات الجزئية لدعم هذه الطريقة. فعلى سبيل المثال، يمكننا تصميم المستويات العليا لبرنامج، كما يلي:

```
procedure Main is
 type Data is...
 procedure Input   ( Item : out   Data) is separate;
 procedure Process ( Item : in out Data) is separate;
 procedure Output  ( Item : in    Data) is separate;
Begin -- Main
...
End Main;
```

وبالفعل، فقد صرحنا عن أرومات لكل إجرائية، والتي يمكن ترجمتها منفصلاً، كما يلي:

```
separate (Main)
procedure Input (Item : out Data) is
Begin
...
End Input;
separate (Main)
procedure Process(Item : in out Data) is
Begin
...
End Process;
separate (Main)
procedure Output (Item : in Data) is
Begin
...
End Output;
```

فكلما أجرينا تحليلاً أكثر لكل وحدة من هذه الوحدات، أمكننا التصريح عن وحدات جزئية أُخرى.

وطريقة التصميم «من الأعلى للأدنى» تعتبر تقليدية لأكثر النظم ولغة ADA تدعم هذه الطريقة. وعلى أي حال، بما أنّ كل وحدة جزئية يجب أن تصرّح عن أصلها، فهذه الطريقة غير صالحة، إذا احتاجت عدة وحدات برمجية استخدام تسهيلات نفس الوحدة.

التطوير من الأدنى للأعلى (Bottom_Up Development):

تسمح هذه الطريقة بخلق وحدات، يمكن أن تتشارك بها عدة وحدات. وفي ADA، قد تمّ استخدام الوحدات المكتبية لتنفيذ هذه الطريقة. ووفق هذه الطريقة، نخلق في البدء الحزم البرمجية والبرامج الجزئية، التي تقدم التسهيلات الأساسية التي نحتاجها، ومن ثم، نبني النظام اعتباراً من الأدوات المقدمة. فعلى سبيل المثال، يمكننا التصريح عن حزمة تصدر أنواع معطيات مجردة Complex.Number، كما يلي:

```
package Complex is
 type Complex is
 type Number is...

... -- abstract operations for Number objects
End Complex;
```

ويمكننا الرجوع لهذه الوحدة في توصيف السياق، ومن ثمّ نستخدم التسهيلات التي تقدمها هذه الحزمة:

```
with Complex;
procedure Main is
...
 My_Number : Complex.Number;
...
Begin
...
End Main;
```

وفي النظم الضخمة، يمكن استخدام كلا الطريقتين «التطوير من الأعلى للأدنـى» و«لتطوير من الأدنى للأعلى» بشكل طبيعي. وفي طريقة التصميم غرضي التوجه التي اقترحناها، نستخدم في الواقع، كلا الطريقتين: حيث نحدد و ننقح الأغراض و العمليات «من الأعلى للأدنى»، ولكن نستخدم تسهيلات الأغراض «من الأدنى للأعلى».

ولنعتبر تمثيل ارتباطات وحدات الترجمة، لنظام على شكل بيان موجه. فإذا اعتمدت الوحدة A على الوحدة B، سيكون هنالك سهم منA إلى B. وفي الطريقة " من الأعلى للأدنى"، فإن عندنا بيانات مرتبطة تشكل شجرة، مع البرنامج الرئيسي كجذر، والشجرات الجزئية (الإجرائيات والوظائف)، تشكل مستويات أوسع وأوسع، كما تمّ تحليلهما. وفي الطريقة غرضية التوجه، فإن هكذا بيانات، تشكل قالباً أساسياً. ومرة ثانية، يكون البرنامج الرئيسي في القمة. وأما في الأدنى، فهناك قليل من الوحدات التي نستطيع إعادة استخدامها بشكل عالي، والمجردة بشكل حسن، مثل Text_IO, Linked_Lists, Trig_Functions. بينما يكون في الوسط، مستويات تعيين التطبيقات للتجريد.





المسألة الخامسة:

إظهار رأسٍ مرتفع

Heads-Up-Display

تعريف المسألة

تحديد الأغراض

تحديد العمليات

تأسيس الرؤية

تأسيس واجهة التخاطب



كما عرضنا في الفصل الثاني، فإنّ أحد أسباب أزمة البرمجيات، هو أنّ إدارة الأنظمة الضخمة ذات المركبات البرمجية الكثيرة تتجاوز أحيانا قدراتنا العقلية. ولا يمكن لمبرمجٍ واحدٍ ضبط جميع تفاصيل نظامٍ مؤلفٍ من مليون سطر. كما أنّ طلب إجراء صيانةٍ من شخصٍ واحدٍ لبرنامج ما، دون تهديم بنيته، هو أيضاً مهمة أصعب بكثير.

في الفصول الأخيرة، فحصنا مواصفات ADA و درسنا عناصرها بالتفصيل. و أكدنا على الطريقة التي تدخل بها ADA العديد من مبادئ البرمجة الحديثة، مثل التجريد، و إخفاء المعلومات. و هذا ما يجعل من ADA ليس فقط لغة برمجةٍ إضافية، ولكن أبعد من ذلك. فـ ADA تحوي مجموعة من الميزات المتماسكة التي تساعد المطوّرين في إدارة الحلول المعقدة.

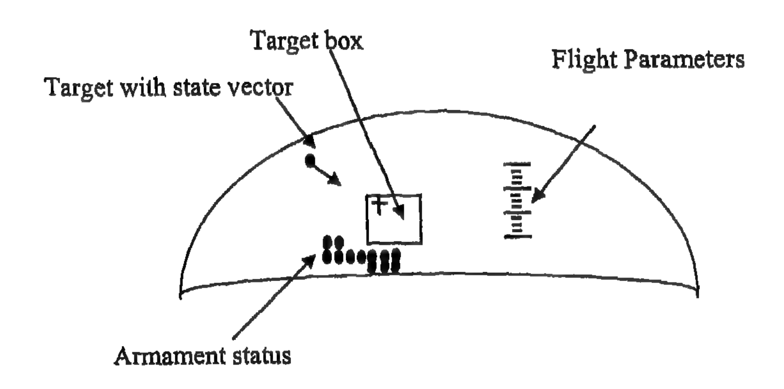
وبالرغم من أنّ فرداً لوحده لا يمكنه دائماً فهم تفاصيل برنامجٍ على غايةٍ من التعقيد، فإنّ ADA تساعد على التعبير عن حلٍ ما بطريقةٍ واضحةٍ ووحدوية (على وحدات Modular). وهكذا، يمكن للمطوّر أن يدرس أجزاء نظامٍ منفصلة، ثم يجمع هذه الأجزاء ليجعل منها كتلةً واحدة. وبالطبع، تنطبق هذه المبادئ على برامج أكثر صغراً أيضاً. وإنّ الهام هو أنّ ADA تساعدنا على التعبير بوضوحٍ عن بنية حلولنا بتقديم وسيلةٍ فعالةٍ للتنفيذ.

وقد درسنا لغاية الآن أربع مسائل ذات تعقيدٍ متزايد. وبما أنّ ADA قد صممت من أجل استخدامها في أنظمةٍ ضخمةٍ جداً، فمن المناسب دراسة الطريقة التي تستخدم بها ADA في تلك الأنظمة. آخذين بعين الاعتبار الحجم المحتمل للحل النهائي (من مرتبة آلاف سطور الترميز)، فلن نقدم النظام كاملاً. وسنطوّر بالأحرى بنيةً عامة لحلّنا، ولن ندرس إلا التجريدات عالية المستوى.

٢١ـ ١ ـ تعريف المسألة (Define the Problem):

خلال معركةٍ جويةٍ لطائرةٍ ذات أداء عالٍ من الهام جداً الحفاظ على واجهة تخاطب رجل/آلة في أبسط ما يمكن. فليس لدينا الوقت لمراجعة (تصفح) لوحة القيادة للحصول على معلومات الطيران. فالطيار يجب أن ينظر إلى الهدف، وبشكل مستمر،

أثناء الاشتباكات القريبة. والحل لهذه المشكلة يتمثل في خلق إظهار رأس عال (HUD)، ليسمح للطيار بملاحظة الهدف و عوامل الطيران الحرجة معاً. وفي أغلب HUD، تكون معلومة الطيران مسقطةً على الفضاء الزجاجي للكبين للسماح للطيار بالنظر إلى خارج طائرته، وبشكل مستمر. يرى الطيار إظهاراً كما هو ممثل في الشكل ٢١ ـ ١.

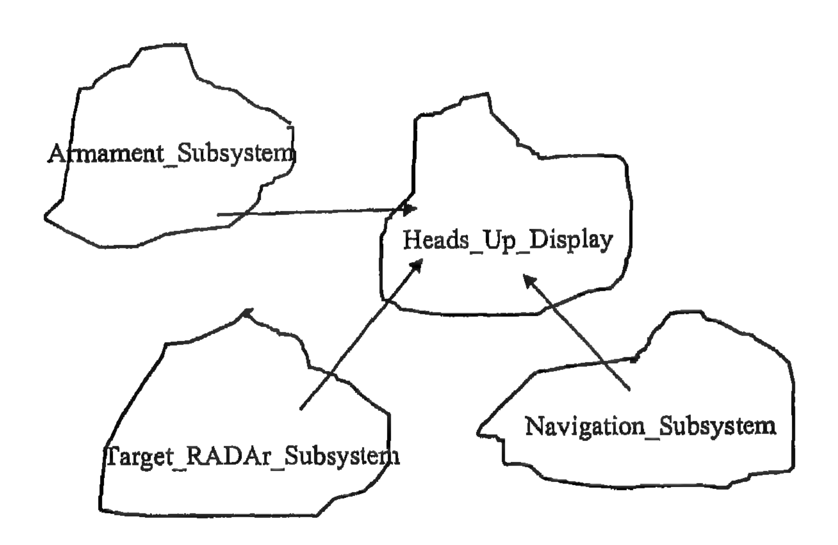


الشكل ٢١ ـ ١. إظهار رأس مرتفع (HUD).

وإن هدف هذا الإظهار، هو تقديم معلومات كافيةٍ بدون تعقيدات كثيرة. ويتمثل السيناريو الصحيح، بالنسبة للطيار، بقيادة الطائرة بطريقةٍ يدخل فيه الهدف المختار إلى منطقة الرمي، حيث أن تنفيذ الرمي في تلك اللحظة، سيعطي احتمالاً كبيراً في إصابة الهدف. وبفرض أنّ الطيار قد اختار مدفعاً مؤتمتاً (أي يمكن التصويب بشكل مؤتمتٍ في حدود بضع درجات)، فإنّ المؤشر يدل على نقطة منطقة الرمي المصوبة حالياً بالمدفع. وأكثر من ذلك، فحسب البعد عن الهدف، وحسب نوع السلاح المختار، فإننا نغير حجم منطقة الرمي لنعين قطر العمل المفيد للسلاح. فعندما تقترب الطائرة من الهدف يزداد حجم المنطقة. ويمثل الإظهار أيضاً العوامل الحرجة للطيران مثل الارتفاع، وزاوية الهجوم بالإضافة إلى حالة السلاح. وأخيراً نركب السهم فوق

الهدف المختار، وهذا لا يؤكد فقط للطيار أنّ هذا هو الهدف المضبوط الملاحق، ولكن تشير له أيضاً إلى اتجاه الطيران المتوقع للهدف.

وفي أنظمةٍ معقدةٍ مثل الطائرات يوجد العديد من الأنظمة الجزئية المحمولة. فعندما نصمم HUD، توجد بالطبع قيود فيزيائية على الحل لا يمكن السيطرة عليها، وبالتالي من غير الممكن إعادة تعريفٍ كاملٍ للطائرة بدلالة احتياجاتنا. والأكثر احتمالاً، وحتى قبل بداية تصميم نظامنا الخاص، يكون فريق تصميم آخر قد عمل مخصصاتٍ وظيفيةٍ للأنظمة الجزئية الأساسية للطائرة، مثل الحاسوب المحمول، أو النظام الجزئي للرادار. وبالنتيجة، يتعلق HUD بواجهات تخاطب متعددة مسبقة التعريف تكون بشكل عام ستاتيكية. وبالرغم من كل هذا يمكن استخدام طريقة التصميم غرضي التوجه من أجل المسألة.



الشكل ٢١ ـ ٢ تخصيص النظام الجزئي في إظهار الرأس المرتفع.

يبين الشكل ٢١ ـ ٢، المخصصات الوظيفية لأنظمة الطائرة من وجهة نظر HUD. الأنظمة الجزئية الأخرى التي يجب أن تتفاعل معهاHUD هي:

١ ـ النظام الجزئي للسلاح: مراقبة موارد الأسلحة و الأهداف.

٢ ـ النظام الجزئي للملاحة: يتضمن جميع التجهيزات المحمولة من أجل قيادة ومراقبة الطائرة.

٣ ـ النظام الجزئي هدف رادار: يلتقط و يتابع الأهداف المتحركة.

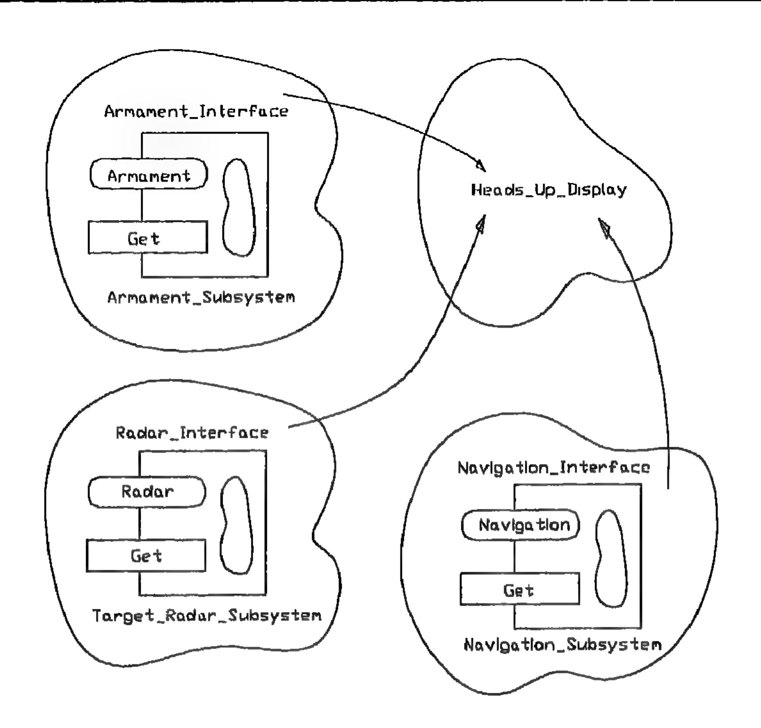
لقد فرض علينا التصميم عالي المستوى، فالأغراض و صفوف الأغراض في هذا المستوى تحوي الأنظمة الجزئية الثلاثة، وحتى HUD نفسه. الآن لن نهتم بتنفيذ أيٍّ من هذه الأغراض، ولكن يلزمنا بالفعل تعريف دقيق لواجهات تخاطب الأغراض التي لا يمكن وصفها إلا بواسطة الحزم البرمجية بـ ADA. بشكل خاص سنفترض أننا نملك أغراضاً خارجية مع عملياتها الموافقة، وهي:

* Armement_SubSystem.Armement_Interface
Type: Armament

Operation: Get

* Navigation_SubSystem.Navigation_Interface
Type: Navigation
Operation: Get
* Target_RADAr_SubSystem.RADAr_Interface
Type: RADAr
Operation: Get

وكل نظامٍ جزئي مثلاً، سيتألف حتماً من عدة وحدات ترجمة. ولسنا بحاجةٍ لمعرفة جميع التفاصيل عن مجموعة النظام الجزئي. وكذلك، سنستخدم في كلٍ من هذه الحالات واجهة تخاطبٍ، وكأنها حزمة برمجية ضمن نظام أوسع، فيمكن استخدام ترميز مركب مختار حتى لا نرجع إلا إلى هذا القسم من النظام الجزئي الكامل. أو يمكن الإستعانة بالبيئة البرمجية للحفاظ على التجريدات منفصلة. وفي كل الأحوال، فإنّ هذه الطرق تراقب التشكيل لواجهة التخاطب، وتعرّف بوضوحٍ مسؤوليات فرقاء التطوير، لكلٍ من الأنظمة الجزئية والـ HUD. ويمكن تمثيل واجهات التخاطب هذه بالشكل ٢١ ـ ٣.



شكل ٢١ - ٣. تصميم واجهة نظام إظهار رأس مرتفع

ونعرّف فيما بعد، واجهات التخاطب هذه بلغة ADA. في هـذا المثـال، سـوف نهمل تفاصيل التمثيليات لكل نوعٍ وسنضيفها عند الحل الكـامل. وبشـكل خـاص، إذا كان كل غرضٍ يمثل تسجيلة معطياتٍ مرسلةً فيما بين نظامين معلوماتيين، فإنه يمكننا استخدام توصيفات تمثيل ADA لوصف تركيب كل مركبٍ من التسجيلة بشـكلٍ واضـح على مستوى الخانة الثنائية (Bit).

ولاحـظ أنّ حزمـةً برمجيـةً كهـذه لواجهـات تخـاطب تدعـم مباشـرةً مبـادئ الوحدوية، والتجريدات التي عرضت في الفصل ٢.

أما داخل نظام السلاح الجزئي، فيمكن تنفيذ واجهة التخاطب، كما يلي:

```
Package Armament_Interface is
 Type Armament is private;
 procedure Get (The_Status: Out Armament);
 private
```

```
 Type Armament is.....
end Armament_Interface;
```

ويوجد واجهات تخاطبٍ مشابهة من أجـل النظـام الجزئـي للملاحـة، والنظـام الجزئي لهدف رادار.

```
Package Navigation_Interface is
 Type Navigation is private;
 procedure Get (The_Status: Out Navigation);
 private
 Type Navigation is.....
end Navigation _Interface;

Package RADAr_Interface is
 Type RADAr is private;
 procedure Get (The_Status: Out RADAr);
 private
 Type RADAr is.....
end RADAr _Interface;
```

ففي هذه الحالات الثلاث، لا يهمنا تطوير أجسام الحزمة البرمجية، لكن يعتبر مسؤولية كل فريقٍ من الفرقاء المكلفين بالأنظمة الجزئية.

لقد وصفنـا الآن، الأغـراض الهامـة للعـالم الخـارجي. وتتمثـل المرحلـة التاليـة بفحص البنية الداخلية لنظامنا الجزئي الخاص.

٢١ ـ ٢ ـ تحديد الأغراض (Identify the Objects):

كمتابعةٍ لطريقـةٍ غرضيـة التوجـه نهتـم الآن بالنظـام الجزئـي لـ HUD نفسـه، ولنفكر بالأغراض التي تهمنا في فضاء المسألة. فمن أجل الوضوح و الصيانة مـن الهـام جداً أن يوافق حلّنا مباشرةً العالم الحقيقي. وفي هذا المستوى، فإن أفضل رؤية ممكنـة لبنية حلّنا هي المقدمة من عيون الطيار. ونستخلص إذاً كـل كيانـات الشـكل ٢١ ــ ١ وكأنها أغراض، وبالتالي يمكن تضمين الكيانات التالية:

- Actual Target (الهدف الحقيقي).
- Armament Status (حالة السلاح).
- Flight Parameters (عوامل الطيران).
- Target Box (منطقة الرمي).

و بالإضافة لذلك، يتم تضمين جهـاز الإظهـار الفيزيـائي ذاتـه، كغـرضٍ نسـميه Head_Up_Display.

٢١ ـ ٣ ـ تحديد العمليات (Identify the Operations):

تتمثل المرحلة التالية بتوصيف سلوك كل التجريدات محددين بذلـك العمليـات التي يخضع لها كل غرض. وقبل عمل ذلك، يجب أن نعزل تجريداً مشـتركاً إضافيـاً. ويجب أن نستخدم نظاماً متشابهاً للإحداثيات لوصـف حالـة جميـع الأغـراض. و مـن الخطـأ الفـادح مثـلاً، استخدام نظـام إحداثيـات مختلـف مـن أجـل الغـرض Actual Target، والطائرة نفسـها. ولهـذا مـن المفيـد تضميـن حزمـةٍ برمجيـةٍ ندعوهـا World System، لتقديم هذا التماثل. وتتصرف هذه الحزمة البرمجيـة كمجموعـة تصريحـات، وفق ما رأيناه في الفصل ١١.

ونحن لا نطلب من هذا الكيـان تقديـم مجموعـة أنـواعٍ معلبـة، وقـد يكـون مـن الحكمة استخدام أنواعٍ بدائيةٍ لوصف التجريدات. ومع ذلك، لا تتوفـر لدينـا معلومـات كافيـة الآن لتصميمنـا لتجميـد واجهـة التخـاطب هـذه. فنسـتخدم إذاً، بعـض الأشـياء البسيطة، مثل:

```
Package World_System is
 Type Latitude is Private;
 Type longitude is Private;
 Type Altitude is Private;
 Type State_Vector is Private;
 Type Dimension is Private;
 Private
 Type Latitude is new float range -90.0..90.0;
```

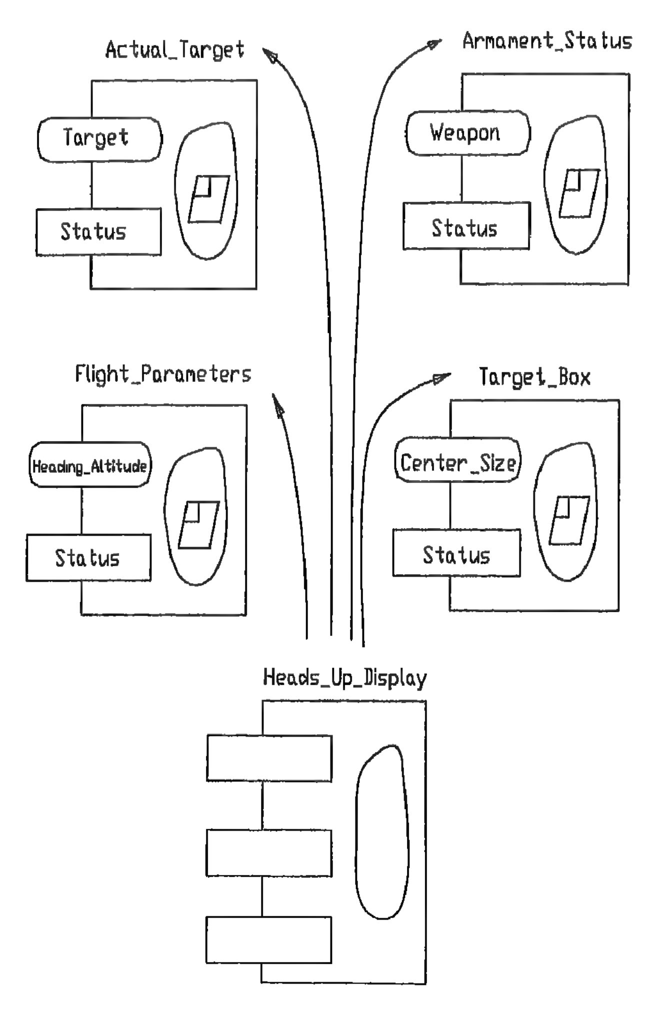
```
 Type longitude is new float range 0.0..360.0;
 Type Altitude is new long_Integer range 0..100_000;
 Type State_Vector is array(1..10) of Boolean;
 Type Dimension is new Integer;
-- the above are only exemples
-- of how these types might be described
end World_System;
```

ويعتبر هذا استخداماً تقليدياً لـ ADA كلغة تصميم مفترضين أننا نقدم تعريفاً كاملاً أصغرياً (مؤقتاً) لكل الأنواع الخاصة هذه. فلقد استخدمنا هذا التوصيف للحزمة البرمجية، لاتخاذ قراراتنا في التصميم دون تجميد أي قرار زرع. ومن الواضح أنّ الحزمة البرمجية هذه المتروكة لوحدها، ستكون غير مفيدة. ولكنها مع ذلك تخدم مقترحنا في هذا المستوى. وخلال تطوير تحليلنا، سنتمم توصيف هذه الوحدة. وسنستخدم استراتيجيةً مماثلةً لتحديد عمليات جميع الأغراض. وفي مستوى التصميم هذا، لن نوصف إلا الرؤية الخارجية الدنيا لكل غرض لتحديد قراراتنا في التصميم. ولهذا، نحن لن نقدم لكل غرض إلا أنواعاً بسيطة، وعمليةً واحدةً لإعطاء حالته. وكل أغراض العالم الحقيقي يمكنها أن تتطور بشكل مواز. ويجب أن ندخل هذه الملاحظة في فضاء حلّنا، ونعالجها و كأنها مهمة.

٢١ ـ ٤ ـ تأسيس الرؤية (Establish the Visibility):

نحن الآن جاهزون لوصف العلاقات بين هذه التجريدات. ففي هذا المستوى من التجريد، يمكن معالجة كل غرض، وكأنّه كيان مستقل تماماً. والاستثناء الوحيد، هو أنّ الغرض HUD، وكما شاهدناه في العالم الحقيقي، يتعلق بجميع الأغراض الأخرى التي حددناها. وبشكل آخر الغرض HUD يجب أن يستورد جميع الأغراض. وتترك لنا هذه الطريقة درجة حريةٍ كبيرة، لأخذ القرار عند التصميم المفصل لكل غرض. وعند تصميم التحليل لكل غرض، في أعلى مستوى، نكون مطمئنين بأنّ التغيرات في غرض ما، لا تؤثر على أي غرضٍ آخر.

والشكل ٢١ ـ ٤، يوضح رؤية هذا التصميم. والقارئ الحريص، سيلاحظ أنّه لا يوجد برنامج رئيسي. وفي الواقع، كما يشير توصيف مسألتنا، فإنّ النظام الجزئي HUD، ليس هو بالضرورة جذر النظام كله. وقد يكون من الأفضل، بالنتيجة، ألا نجّرده إلا على شكل حزمٍ برمجية. فماذا حدث للأنظمة الجزئية المختلفة، التي عزلناها سابقاً. إنها ليست هامة في هذا المستوى من التجريد. ونفترض بالأحرى، أننا بحاجة إليها لزرع جميع الأغراض التي وضعناها. غير أنّه كان من الهام تحديد واجهات التخاطب هذه في مرحلة مبكرة لتحديد فضاء مسألتنا.



الشكل ٢١ ـ ٤. تصميم نظام إظهار رأس مرتفع.

٢١ ـ ٥ ـ تأسيس واجهة التخاطب

(Establish The Interface):

نتابع مع تأسيس قراراتنا في التصميم على شكل توصيف الحزم البرمجية ADA. ومرة أُخرى، لدينا القليل من التفاصيل لعرضها في هذا المستوى من التصميم. ومن المفيد مع ذلك، متابعة هذه المرحلة حتى تسمح لنا باتخاذ قراراتنا لبنية حلّنا، ومن أجل بقية المطوّرين و مدراء البرامج. وبما أننا نملك المعلومات، فإننا نلتقط الرؤية الخارجية لجميع التجريدات، كما يلي:

```
With World_System;
Package Actual_Target is
 Type Target is New World_System.State_Vector;
 Task Coupler is
 entry Status(The_Target: Out Target);
 end Coupler;

With World_System;
Package Armament_Status is
Type Weapon is (Fixed_Gun, Missile);
Task Coupler is
 entry Status(The_Weapon:Out Weapon;The_Quantity:OutNatural);
end Coupler;
end Armament _ Status;

With World_System;
Package Flight_Parameters is
Type Altitude is new World_System.Altitude;
Type Heading is new World_System.State_Vector;
Task Coupler is
 entry Status(The_ Altitude: Out World_System.Altitude; The_Heading:
Out
 World_System.Stat_Vector);
end Coupler;
```

```
end Flight _ Parameters;
Package Target_Box is
 Type Center is new World_System.State_Vector;
 Type Size is new World_System.Dimension;
 Task Coupler is
 entry Status(The_ Center: Out World_System. State_Vector; The_ Size: Out World_System.
 Dimension);
 end Coupler;

 end Target _ Box;
Package Heads_Up_Display
 Task Coupler is
 entry Start;
 end Coupler;
end Heads_Up_Display;
```

ولا يدخل المستوى التالي لحلّنا تصميماً جديداً، ولذلك، لن نتابع. واعتبر مع ذلك، أننا قد حصلنا عليه. فلدينا الآن نظام متوافق بنيته متوافقة مباشرة، مع فهمنا لفضاء المسألة.

وأكثر من ذلك فقد رأينا كيف يمكن استخدام ADA للتصدي لمسائل هامة: فنحدد خارجياً واجهات التخاطب، ونبين النظم الجزئية بفضل وحدات ADA المترجمة بشكل منفصل. وبهذه الطريقة، لا نفصل فقط كحدٍ أقصى، جميع الأنظمة الجزئية، ولكن نقلل كذلك تعقيد الرؤية الخارجية لكل من الأنظمة الجزئية. أي أخذنا معاني ADA من أجل الوحدات الصغيرة (أي حزم برمجية صغيرة) و طبقناها على تجريداتٍ أوسع بكثير (أنظمة جزئية كاملة). ويمكن الاستنتاج إذاً، أنَّ مفاهيم ADA، تسمح على السواء بالتحكم بتعقيد الأنظمة المعقدة، وبالتجريدات الصغيرة.

❁ ❁ ❁





واصفات اللغة مسبقة التعريف

Predefined Language Attributes



إن هـــذا الملحـــق مـــأخوذ، بالســـماح مـــن مكتـــب البرمجـــة المشـــترك بـــــ

(REDSUO) “eciffO margorP tnioJ ADA”

"egugnaL gnimmargorP ADA eht rof launaM ecnerefeR"

P’Address	من أجل سـابقة P تشير لغـرض، لوحـدة برمجيـة، لعلامة، أو لكيان: تعيد عنوان أول وحدة تخزين محجوزة لـP. من أجل برنـامج جزئـيٍ، حزمـة برمجيـة، وحـدة مهمـة أو علامة، ترجع هذه القيمـة إلـى ترميز الآلـة المتعلـق بالجسم أو بالتعليمة الموافقـة. مـن أجـل مدخـل تـمّ إعطـاء عبـارة عنـوان، ترجـع القيمـة إلـى المقاطعـة الصلبة الموافقة. قيمـة هـذا الواصـف هـي مـن النـوع Address والمعرف في الحزمة البرمجية System.
P'Aft	من أجل سابقة P تشير لنوعٍ جزئـيٍ ممثـلٍ بالفاصلـة الثابتة: تعيـد عـدد الأرقـام العشرية الضرورية بعـد الفاصلـة العشرية لتفرض دقـة النـوعِ الجزئـيِ P، إلا إذا كـان مميز P أكبر من 0.1، في هكذا حالة يعيـد الواصـف القيمة 1 (P'Aft يمثل أصغر عدد طبيعـي موجـب N الـذي مـن أجلـه (10**N)*P'Delta أكـبر أو تسـاوي واحـد). قيمـة هـذا الواصـف هـي مـن النـوعِ *Universal Integer*.
P'Base	من أجـل سابقة P تشير لنوعٍ أو لنوعٍ جزئيٍ: يشـير هـذا الواصـف للنـوع الأساسـي مـن P، و هـو مباح فقط كسـابقة لاسم واصفٍ آخـر، علـى سبيل المثال، P'Base'First.

P'Callable	من أجل سابقة P مطابقة لنوع مهمة: يُرجع القيمة False عندما ينتهي أو يكتمل تنفيذ المهمة P، أو عندما تكون المهمة شاذة؛ يعيد القيمة True في غير ذلك. قيمة هذا الواصف هي من النوع *Boolean.*
P'Constrained	من أجل سابقة P تشير لغرض من نوع مع مميزات: يعيد القيمة True إذا طُبق قيد مميز على الغ[illegible] P، أو إذا كان الغرض ثابتاً (متضمناً معاملاً صورياً أو معاملاً مولداً من النموذج in)؛ يعيد القيمة False في غير ذلك. إذا كان P معاملاً صورياً مولداً من النموذج in out، أو إذا كان P معاملاً صورياً من النموذج in out أو out وإذا كانت علامة النوع المعطي في توصيف المعامل الموافق تشير لنوعٍ غير مقيدٍ مع مميزات، عندها نحصل على قيمة هذا الواصف من المعامل الحقيقي الموافق. قيمة هذا الواصف من النوع *Boolean.*
P'Constrained	من أجل سابقة P تشير لنوعٍ خاص، أو لنوعٍ جزئي خاص: يعيد القيمة False إذا أشار P إلى نوع خاص غير صوري ومقيد مع مميزات؛ يعيد أيضاً القيمة False إذا أشار P إلى نوع مولد صوري خاص وإذا كان النوع الجزئي الحقيقي المرتبط يتمثل إما بنوع غير مقيد مع مميزات أو إما بنوع مصفوفة غير مقيدة؛ ويعيد القيمة True في غير ذلك. قيمة هذا الواصف من النوع *Boolean.*

P'Count	من أجل سابقة P تشير لمدخل من وحدة مهمة: يعيد العدد الحالي لاستدعاءات مدخل موجودة في رتل انتظار المدخل (إذا تمّ تقييم الواصف داخل تعليمة accept من أجل المدخل P، عندها Count لا يتضمن المهمة الداعية). قيمة هذا الواصف من النوع *universal Integer.*
P'Delta	من أجل سابقة P تشير لنوعٍ جزئيٍ ممثلٍ بالفاصلة الثابتة: يعيد قيمة المميز المخصص في تعريف الدقة الثابتة للنوع الجزئي .P قيمة هذا الواصف من النوع *Universal Real.*
P'Digits	من أجل سابقة P تشير لنوعٍ جزئيٍ ممثلٍ بالفاصلة العائمة: يعيد عدد الأرقام العشرية في الجزء العشري لأعداد نموذجية من النوع الجزئي .P قيمة هذا الواصف من النوع *univerasl_Integer.*
P'Emax	من أجل سابقة P تشير لنوعٍ جزئيٍ ممثلٍ بالفاصلة العائمة: يعيد قيمة أكبر أس في الشكل القانوني الثنائي لأعداد نموذجية من النوع الجزئيP. (هذا الواصف يعيد الناتج 4*B) قيمة هذا الواصف من النوع *univerasl Integer.*

P'Epsilon	من أجل سابقة P تشير لنوعٍ جزئـيٍ ممثـلٍ بالفاصلـة العائمة: يعيد القيمة المطلقة للفرق بين العـدد النمـوذج 1.0 و العدد النمـوذج الـذي يليـه مباشـرة مـن أجـل النـوع الجزئـي P. قيمـــة هـــذا الواصـــف مـــن النـــوع *universal real.*
P'First	من أجل سابقة P تشير لنـوعٍ جزئـيٍ سـلمي أو لنـوعٍ جزئيٍ من نوع سلمي: يعيد الحد الأدنى من P. قيمة هذا الواصف لها نفس النوع P.
P'First	من أجـل سـابقة P مطابقـة لنـوعٍ مصفوفـة، أو التـي تشير لنوعٍ جزئيٍ لمصفوفة مقيدة: يعيد قيمة الحد الأدنى من أول دليـل المجـال لـ P. قيمة هذا الواصف لها نفس نوع الحد الأدنى هذا.
P'First(N)	من أجـل سـابقة P مطابقـة لنـوعٍ مصفوفـة، أو التـي تشير لنوعٍ جزئيٍ لمصفوفة مقيدة: يعيد قيمة الحد الأدنى لدليل المجـال رقـم N. قيمـة هذا الواصف لها نفس نوع الحد الأدنـى هـذا. يجـب أن يكـون المعـامل N تعبـيراً سـاكناً مـن النـوع *univerasl_Integer.* يجب أن تكون قيمة N موجبـة و غير معدومة و ليست أكبر من بعد المصفوفة.
P'First_Bit	من أجل سابقة P تشير لمركب من غرض تسجيلة: يعيد الإنزياح بين بداية أول وحدة تخزيـن محجـوزة مـن قبـل المركـب و أول خانـة ثنائيـة يحجزهـا المركب. يقاس هذا الإنزياح بالخانات الثنائية. قيمـة هذا الواصف من النوع *univerasl_Integer.*

P'Fore	من أجل سابقة P تشير لنوعٍ جزئـيٍ ممثـلٍ بالفاصلـة الثابتة : يعيـد أصغـر عـدد مـن المحـارف الضروريـة للقسـم الطبيعي مـن التمثيـل العشـري لأي قيمـة مـن النـوع الجزئـي P، مفـترضين أنّ التمثيـل لا يحتـوي أسـاً، لكنه يتضمن سابقة مؤلفة من محرف واحد إما إشـارة الناقص أو فراغ. (هـذا العـدد ليـس أصغـر مـن 2، و لا يحتـوي أي أصفار غير ضرورية أو رموز من الشـكل "_"). قيمـة هذا الواصف من النوع *univerasl_Integer.*
P'Image	من أجل سابقة P تشير لنوعٍ متقطـع أو لنـوعٍ جزئـيٍ متقطع : يمثل هذا الواصف تابعاً بمعاملٍ واحدٍ فقط. يجب أن يمثل المعامل الفعلي X قيمة من النوع الأساسي لـ P. نـوع النتيجـة هـو مـن النـوع المعـرف مسـبقاً String والنتيجة هي صورة للقيمـة X، أي هـي سلسـلة مـن المحارف الممثِلة للقيمـة فـي شـكل الإظهـار. صـورة قيمة صحيحة تتمثل بالمحـارف العشـرية الموافقـة، بدون رموز من الشكل "_"، ولا أصفـار فـي الـرأس، ولا أس، ولا فراغات متدلية، لكن مع حـرفٍ واحـدٍ في الرأس كسابقة يمثل إشارة الناقص أو فراغ. تمثل صورة قيمة مرقمة إما المُعرف الموافـق بـأحرف كبـيرة أو المحـارف الرمزيـة الموافقـة بمـا فـي ذلـك الفـاصلتين العلويتيـن؛ لا يتضمـن ذلـك الفراغـات المتدلية و لا الفراغات الرأسية. صـورة محـرف غـير بياني يُعرّف من التنفيذ.

P'Large	من أجل سابقة P تشير لنوعٍ جزئيٍ حقيقي: يعيد هذا الواصف أكبر عدد نموذج موجب من النوع الجزئي .P قيمة هذا الواصف من النوع *universal real.*
P'Last	من أجل سابقة P تشير لنوعٍ جزئيٍ سلمي: يعيد الحد الأعلى من .P قيمة هذا الواصف لها نفس نوع .P
P'Last	من أجل سابقة P مطابقة لنوعٍ مصفوفة، أو التي تشير لنوعٍ جزئيٍ لمصفوفة مقيدة: يعيد قيمة الحد الأعلى من أول دليل المجال. قيمة هذا الواصف لها نفس نوع الحد الأعلى هذا.
P'Last(N)	من أجل سابقة P مطابقة لنوعٍ مصفوفة، أو التي تشير لنوعٍ جزئيٍ لمصفوفة مقيدة: يعيد هذا الواصف قيمة أعلى حد من دليل المجال رقم .N قيمة هذا الواصف لها نفس نوع الحد الأعلى هذا. يجب أن يكون المعامل N تعبيراً ساكناً من النوعٍ *univerasl_Integer.* يجب أن تكون قيمة N موجبة و غير معدومة و ليست أكبر من بعد المصفوفة.
P'Last_Bit	من أجل سابقة P تشير لمركب من غرض تسجيلة: يعيد الإنزياح بين بداية أول وحدة تخزين محجوزة من قبل المركب و آخر خانة ثنائية يحجزها المركب.يقاس هذا الإنزياح بالخانات الثنائية. قيمة هذا الواصف من النوعٍ *univerasl_Integer.*

P'Length	من أجل سابقة P مطابقة لنوعٍ مصفوفة، أو التي تشير لنوعٍ جزئيٍ لمصفوفة مقيدة: يعيد عدد القيم لأول دليل المجال (صفر من أجل مجال فارغ). قيمة هذا الواصف من النوع *univerasl Integer*.
P'Length(N)	من أجل سابقة P مطابقة لنوعٍ مصفوفة، أو التي تشير لنوعٍ جزئيٍ لمصفوفة مقيدة: يعيد عدد القيم لدليل المجال رقم N (صفر من أجل مجال فارغ). قيمة هذا الواصف من النوع *univerasl_Integer*. يجب أن تكون قيمة N موجبة و غير معدومة و ليست أكبر من بعد المصفوفة.
P'Machine_Emax	من أجل سابقة P تشير لنوعٍ جزئيٍ ممثلٍ بالفاصلة العائمة: يعيد أكبر قيمة للأس لتمثيل الآلة من النوع الأساسي من P. قيمة هذا الواصف من النوع *univerasl_Integer*.
P'Machine_Emin	من أجل سابقة P تشير لنوعٍ جزئيٍ ممثلٍ بالفاصلة العائمة: يعيد أصغر قيمة (أصغر قيمة سالبة) للأس لتمثيل الآلة من النوع الأساسي من P. قيمة هذا الواصف من النوع *univerasl_Integer*
P'Machine_Mantissa	من أجل سابقة P تشير لنوعٍ أو لنوعٍ جزئيٍ ممثلٍ بالفاصلة العائمة: يعيد عدد الأرقام في القسم العشري لتمثيل الآلة للنوع الأساسي من P (الأرقام تمثل أرقام معممة على المجال [0..P'Machine_Radix-1]. قيمة هذا الواصف من النوع *univerasl_Integer*.

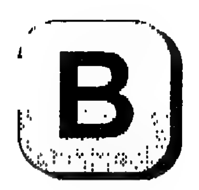
P'Machine_Overflows	من أجل سابقة P تشير لنوعٍ أو لنوعٍ جزئيٍ حقيقي: يعيد القيمة True إذا كانت جميع العمليات المسبقة التعريف على قيم من النوع الأساسي من P تقدم نتيجة صحيحة أو تبرز الاستثناء Numeric_Error في حالات الطوفان، تعيد القيمة False في غير ذلك. قيمة هذا الواصف من النوع *Boolean.*
P'Machine_Radix	من أجل سابقة P تشير لنوعٍ أو لنوعٍ جزئي ممثلٍ بالفاصلة العائمة: يعيد قيمة الأساس المستخدم بواسطة تمثيل الآلة من النوع الأساسي من P. قيمة هذا الواصف من النوع *univerasl Integer.*
P'Machine_Rounds	من أجل سابقة P تشير لنوعٍ أو لنوعٍ جزئي حقيقي: يعيد القيمة True إذا كانت جميع العمليات الحسابية المسبقة التعريف على قيم من النوع الأساسي من P تعيد قيمة دقيقة أو نتيجة مدورة، تعيد القيمة False في غير ذلك. قيمة هذا الواصف من النوع *Boolean.*
P'Mantissa	من أجل سابقة P تشير لنوعٍ جزئي حقيقي: يعيد عدد الأرقام الثنائية في الجزء الثنائي من أعداد نموذجية من النوع الجزئي P. (هذا الواصف يعيد العدد B من أجل التمثيل بالفاصلة العائمة أو التمثيل في الفاصلة الثابتة). قيمة هذا الواصف من النوع *univerasl Integer.*
P'Pos	منأجل سابقة P تشير لنوعٍ متقطع أو لنوعٍ جزئيٍ متقطع: يمثل هذا الواصف تابعاً بمعامل واحد. يجب أن

	يمثل المعامل الفعلي X قيمة من النوع الأساسي من P. النتيجة من النوع *universal_Integer* ؛ يمثل رقم الموضع لقيمة المعامل الفعلي.
P'Position	من أجل سابقة P تشير لمركب من غرض تسجيلة : يعيد الإنزياح بين بداية أول وحدات التخزين التي تحجزها التسجيلة و أول وحدات التخزين التي يحجزها المركب. يقاس هذا الإنزياح بوحدات التخزين. قيمة هذا الواصف من النوع *univerasl Integer*.
P'Pred	من أجل سابقة P تشير لنوعٍ متقطع أو لنوعٍ جزئي متقطع : يمثل هذا الواصف تابعاً بمعامل واحد. يجب أن يمثل المعامل الفعلي X قيمة من النوعِ الأساسي من P؛ النتيجة تمثل القيمة التي موضعها يسبق مباشرة موضع X. يبرز الاستثناء Constraint_Error إذا كانت العلاقة X=P'Base'First محققة.
P'Range	من أجل سابقة P مطابقة لنوعٍ مصفوفة، أو التي تشير لنوعٍ جزئي لمصفوفة مقيدة: يعيد أول دليل المجال من P، و الذي يكون، المجال P'First.. P'Last.
P'Range(N)	من أجل سابقة P مطابقة لنوعٍ مصفوفة، أو التي تشير لنوعٍ جزئي لمصفوفة مقيدة: يعيد الدليل رقم N من P، و الذي يكون، المجال P'First(N).. P'Last(N).
P'Safe_Emax	من أجل سابقة P تشير لنوعٍ جزئيٍ ممثلٍ بالفاصلة العائمة :

يعيد قيمة أكبر أس في الشكل القانوني الثنائي من الأعداد الأكيدة من النوع الأساسي من .P (يعيد هذا الواصف العدد E). قيمة هذا الواصف من النوع *univerasl Integer.*	
من أجل سابقة P تشير لنوعٍ جزئي حقيقي: يعيد أكبر عدد أكيد موجب من النوعِ الأساسي من .P قيمة هذا الواصف من النوع *universal_real.*	P'Safe_Large
من أجل سابقة P تشير لنوعٍ جزئي حقيقي: يعيد أصغر عدد أكيد موجب (غير معدوم) من النوع الأساسي من .P قيمة هذا الواصف من النوع *universal real.*	P'Safe_Small
من أجل سابقة P تشير لغرض: يعيد عدد الخانات الثنائية المحجوزة لتحتوي الغرض. قيمة هذا الواصف من النوع *univerasl Integer.*	P'Size
من أجل سابقة P تشير لأي نوع أو نوع جزئي: يعيد العدد الأصغري من الخانات الثنائية التي يحتاجها التنفيذ لاحتواء أي غرض ممكن من النوع أو النوع الجزئي .P قيمة هذا الواصف من النوع *univerasl Integer.*	P'Size
من أجل سابقة P تشير لنوعٍ جزئي حقيقي: يعيد أصغر عدد نموذج موجب (غير معدوم) من النوع الجزئي .P قيمة هذا الواصف من النوع *universal real.*	P'Small
من أجل سابقة P تشير لنوعٍ وصول أو لنوعٍ جزئي للوصول: يعيد العدد الكلي من وحدات التخزين المحجوزة	P'Storage_Size

	للمجموعة المرتبطة مع النوع الأساسي من .P قيمة هذا الواصف من النوع *univerasl_Integer*.
P'Storage_Size	من أجل سابقة P تشير لنوعٍ مهمة أو لغرض مهمة: يعيد عدد وحدات التخزين المحجوزة لكل تنشيط لمهمة من النوع P أو لتنشيط غرض المهمة .P قيمة هذا الواصف من النوع *univerasl_Integer*.
P'Succ	من أجل سابقة P تشير لنوعٍ متقطع أو لنوعٍ جزئي متقطع: يمثل هذا الواصف تابعاً بمعامل واحد. يجب أن يمثل المعامل الفعلي X قيمة من النوع الأساسي من .P النتيجة من النوع الأساسي من P ؛ النتيجة تمثل القيمة التي موضعها يلي مباشرة موضع .X يبرز الاستثناء Constraint_Error إذا كانت العلاقة X=P'Base'Last محققة.
P'Terminated	من أجل سابقة P تشير لغرض مهمة: يعيد القيمة True إذا انتهت المهمة P ؛ يعيد القيمة False في غير ذلك. قيمة هذا الواصف من النوع *Boolean*.
P'Val	من أجل سابقة P تشير لنوعٍ متقطع أو لنوعٍ جزئي متقطع: يمثل هذا الواصف تابعاً خاصاً بمعامل واحد و الذي يمكن أن يكون من أي نوع صحيح. تكون النتيجة من النوع الأساسي من P ؛ و هي القيمة التي رقم موضعها يمثل القيمة الصحيحة الموافقة للمعامل الفعلي. يبرز الاستثناء Constraint_Error إذا لم تنتم قيمة X للمجال P'Pos(P'Base'First)..P'Pos(P'Base'Last)

P'Value	من أجل سابقة P تشير لنوعٍ متقطـع أو لنـوعٍ جزئـي متقطع: يمثل هـذا الواصـف تابعـاً بمعـامل واحـد. يجـب أن يكـون المعـامل الفعلـي X قيمـة مــن النــوع String المسـبق التعريـف. نـوع النتيجـة يكـون مــن النــوع الأساسي من .P يهمل أي فـراغ ترويسـة أو أي فـراغ متدلي من سلسلة المحارف الموافقة لـ .X من أجل نوع مرقم، إذا كانت سلسلة المحــارف لهـا دلالة حرف مرقم، و إذا وجد هذا الحرف مــن أجـل القاعدة الأساسية منP، عندها تطـابق النتيجـة قيمـة مرقمة. من أجل النــوع الصحيـح، إذا كـانت سلسـلة المحارف لها دلالة حرف صحيح مع حرف ترويسـة وحيد اختيـاري و الـذي يمثـل إشـارة + أو -، و إذا وجدت قيمة موافقــة فـي القـاعدة الأساسية مـن P، تكون النتيجة هي هذه القيمة. في أي حالة أُخـرى، سيبرز الاستثناء .Constraint_Error
P'Width	من أجل سابقة P تشير لنوعٍ جزئي متقطع: يعيد الطول الأعظمي للصور من أجل جميع قيم النوع الجزئي P (تتمثل الصورة بسلسلة المحــارف المُعـادة بواسـطة الواصـف Image). قيمـة هـذا الواصـف مـن النوع *universal_Integer*.



عمليات اللغة مسبقة التعريف

Predefined Language Pragmas



إن هذا الملحق مأخوذ، بالسماح من مكتب البرمجة المشتركة بـ

"ADA Joint Program Office" (OUSDER)

من وزارة الدفاع الأمريكية من:

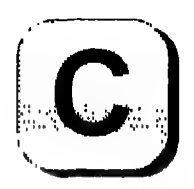
"Reference Manual for the ADA Programming Language," (1983), Appendix B.

العملي:	المعنى:
Controlled	يأخذ الاسم البسيط لنوع وصول (Access type) كمعاملٍ وحيد. يُخصص هذا العملي مباشرة داخل قسم التصريح أو توصيف حزمة برمجية التي تحتوي على تصريحٍ من نوع الوصول، يجب أن يحدث التصريح قبل العملي. ولم يُخصص هذا العملي من أجل الأنواع المشتقة. يحدد هذا العملي بأنّ الإصلاح الآلي للذاكرة يجب ألا يُنجز من أجل الأغراض المعينة بقيمٍ من نوع الوصول، ما عدا في حال ترك الأعمق من تعليمة الكتلة، أو جسم برنامجٍ جزئي، أو جسم مهمة، والتي يطوق تصريح نوع الوصول، أو بعد ترك البرنامج الرئيسي.
Elaborate	يأخذ اسماً بسيطاً أو أكثر، مشيراً للوحدات المكتبية كمعاملات. يُخصص هذا العملي مباشرة بعد عبارة السياق من وحدة الترجمة (قبل الوحدة المكتبية أو الوحدة الثانوية التي تلي). يجب أن يكون كل معامل اسماً بسيطاً لوحدةٍ مكتبية مذكوراً بواسطة عبارة السياق. يُحدد هذا العملي بأنّه يجب تطوير جسم الوحدة المكتبية الموافق قبل وحدة الترجمة المعطية. وإذا كانت الوحدة المترجمة تمثل وحدةً جزئيةً، يجب تطوير جسم الوحدة المكتبية قبل جسم الوحدة البرمجية السلف من الوحدة الجزئية.
Inline	يأخذ اسماً بسيطاً أو أكثر كمعاملات؛ كل معامل يمثل إما اسم إجرائية جزئية أو اسم برنامج جزئي مولد. هذا العملي مخصص

	فقط في مكان عنصر مصرح في قسم تصريح أو توصيف حزمة برمجية ؛ أو بعد وحدة مكتبية في الترجمة ، لكن قبل أي وحدة ترجمة تليها. يحدد هذا العملي بأن التوسع بالأسطر لأجسام الإجرائيات الجزئية يجب أن تنتشر فوراً في كل استدعاء ممكن ؛ في حالة البرنامج الجزئي المولد، يطبق العملي في استدعاء نسخه المؤقتة.
Interface	يأخذ اسم لغة و اسم إجرائية كمعاملات. وهذا العملى مخصص فقط في مكان عنصر مصرح، و يجب تطبيقه في هذه الحالة لبرنامج جزئي مصرح عنه بواسطة عنصر تصريح سابق من نفس قسم التصريح أو من نفس توصيف الحزمة البرمجية. هذا العملي مخصص أيضاً من أجل وحدة مكتبية، في هذه الحالة، يجب أن يظهر العملي بعد التصريح عن البرنامج الجزئي، و قبل أي وحدة ترجمة تليها. يحدد العملي اللغة الثانية (ومن ذلك اصطلاحات الاستدعاء) و يُخبر المترجم بأن وحدة غرض ستقدم للبرنامج الجزئي الموافق.
List	يأخذ إحدى المعرّفين On أو Off كمعامل وحيد. وهذا العملي مخصص في كل مكان يخصص فيه عملي. ويحدد هذا العملي بأنّ جدول الترجمة يجب أن يُتابع أو يجب أن يُحذف إلى أن يُعطى عملي جدول مع معامل معاكس داخل نفس الترجمة. دائماً العملي نفسه يجدول إذا كان المترجم ينتج جداول.
Memory_Size	يأخذ حرفاً رقمياً كمعامل وحيد. هذا العملي مخصص فقط في بداية ترجمة، قبل أي وحدة ترجمة (إذا وجد شيء). يتمثل تأثير هذا العملي باستخدام قيمة الحرف الرقمي المحدد من أجل تعريف العدد المسمى Memory_Size .

Optimize	يأخذ أحد المعرّفين Time أو Space كمعامل وحيد. هذا العملي مخصص فقط داخل قسم تصريح و يطبق لكتلة أو لجسم يحتوي قسم التصريح. ويحدد فيما إذا كان الزمن أم الذاكرة يمثل معيار الأمثلية الأساسي.
Pack	يأخذ الاسم البسيط من نوع تسجيلة أو نوع مصفوفة كمعامل وحيد. المواضع المخصصة لهذا العملي، و القيود على اسم النوع ، موجهة بنفس القواعد التي من أجل عبارة التمثيل. يحدد العملي بأنّه يجب أن يكون المعيار الرئيسي عند اختيار التمثيل للنوع المُعطى هو تصغير التخزين.
Page	لا يأخذ هذا العملي أي معامل، و هو مخصص في أي مكان مخصص للعملي. يحدد بأنّ نص البرنامج الذي يلي العملي يجب أن يبدأ على صفحة جديدة (إذا كان المترجم يُنتج حالياً جدول).
Priority	يأخذ تعبيراً ساكناً من النوع الجزئي Integer المسبق التعريف Priority كمعامل وحيد. هذا العملي مخصص فقط داخل توصيف وحدة مهمة أو مباشرة داخل القسم التصريح الأكثر بعداً من برنامج رئيسي. يحدد أفضلية المهمة (أو مهام من نوع المهمة) أو أفضلية البرنامج الرئيسي.
Shared	يأخذ اسماً بسيطاً لمتحول كمعامل وحيد. هذا العملي مخصص فقط من أجل متغير مصرح عنه بواسطة تصريح غرض و هو من نوع سلمي أو من نوع الوصول، يجب أن يحدث كل من تصريح المتغير و العملي (وفق هذا الترتيب) مباشرة داخل نفس قسم التصريح أو نفس قسم توصيف الحزمة البرمجية. يحدد هذا العملي بأنّ كل قراءة أو تعديل من المتغير يمثل نقطة تزامن لذلك المتغير. ويجب على التنفيذ أن يحصر الأغراض التي من أجلها مخصص هذا العملي لأغراض من أجلها تكون كل قراءة أو تعديل مباشر منفذاً كعملية غير مرئية.

Storage_Unit	يأخذ محرفاً رقمياً كمعامل وحيد. هـذا العملي مخصص فقط في بداية ترجمة، قبل أول وحدة ترجمة (إن وجدت) من الترجمة. يتمثل تأثير هـذا العملي باستخدام قيمة الحرف الرقمي المعين لتعريف الرقم المسمى Storge_Unit .
Suppress	يأخذ كمعاملات معرف التحقق و اختيارا اسم غرض، نوع أو نوع جزئي، برنامج جزئي، وحدة مهمة، أو وحدة مولدة. هـذا العملي مخصص فقط إما مباشرة داخل قسم التصريح أو مباشرة داخل توصيف حزمة برمجية. في الحالة الأخيرة، الشكل الوحيد المستخدم يتمثل بالاسم الذي يشير لكيان (أو عدة برامج جزئية محملة زائداً) مصرح عنه مباشرة داخل توصيف الحزمة البرمجية.يمتد السماح بإهمال التحقيق المحدد اعتباراً من مكان العملي و حتى نهاية منطقة التصريح المرتبطة مع أعمق عبارات كتلة أو وحدات برنامج شاملة. من أجل عملي محدد في توصيف حزمة برمجية، يمتد السماح حتى نهاية مدى الكيان المسمى. إذا تضمن العملي اسماً، ينحصر السماح أكثر بإهمال التحقيق المحدد: يُعطى فقط من أجل العمليات على غرض مسمى أو على جميع الأغراض من النوع الأساسي من نوع مسمى أو نوع جزئي مسمى، من أجل استدعاءات برنامج جزئي مسمى، من أجل تنشيطات مهام من نوع المهمة المسماة، أو من أجل النسخ المؤقتة من الوحدة المولدة المحددة.
System_Name	يأخذ محرفاً رقمياً كمعامل وحيد. هذا العملي مخصص فقط في بداية ترجمة، قبل أول وحدة ترجمة (إن وجدت) من الترجمة. يتمثل تأثير هـذا العملي باستخدام الحرف الرقمي مع المُعرف المعين لتعريف الثابت System_Name . أيضاً مخصص هذا العملي فقط إذا وافق المعرف المعين لواحد من الحروف من النوع Name المصرح عنه في الحزمة البرمجية System.



بيئة اللغة مسبقة التعريف

Predefine Language Environment



هذا الملحق مأخوذ، بالسماح من مكتب البرمجة المشتركة بـ
"ADA Joint Program Office" (OUSDER)
من وزارة الدفاع الأمريكية من

"Reference Manual for the ADA Programming Language,"

(1983), Appendix C

يتضمن تعريف لغة ADA عدة وحدات مكتبية مسبقة التعريف. من هذه الوحدات المكتبية مايلي:

- **Text_IO**
- **Low_Level_IO**
- **Calendar**
- **Unchecked_Conversion**
- **Sequential_IO**
- **Direct_IO**
- **Unchecked_Deallocation**
- **System**
- **IO_Exceptions**

بالإضافة لذلك، توجد حزمة برمجية مسبقة التعريف تُسمى Standard و التي تحتوي جميع الأغراض و العمليات مسبقة التعريف. عند ترجمة وحدة مكتبية، يعالج المترجم الوحدة و كأنّه مصرح عنها في نهاية توصيف الحزمة البرمجية Standard . و بالتالي، جميع هذه المعرفات مسبقة التعريف تكون مرئية مباشرة.

نجد فيما بعد توصيف الحزمة البرمجية Standard الجسم لم يعرض لأنه يتعلق بالتنفيذ. يشير الرمز {...} لقيمة معرف تنفيذها. توصيف بقية الوحدات البرمجية مسبقة التعريف يتبع ذلك من الحزمة البرمجية StanADArd .

STANDARD

```
package STANDARD is
  type BOOLEAN is (FALSE, TRUE);
  function "="   (LEFT, RIGHT : BOOLEAN) return OOLEAN;
  function "/="  (LEFT, RIGHT : BOOLEAN) return OOLEAN;
  function "<"   (LEFT, RIGHT : BOOLEAN) return OOLEAN;
  function "<="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  function ">"   (LEFT, RIGHT : BOOLEAN) return OOLEAN;
  function ">="  (LEFT, RIGHT : BOOLEAN) return BOOLEAN;
  function "not" (X           : BOOLEAN)return BOOLEAN;
  function "and" (X, Y        : BOOLEAN) return BOOLEAN;
  function "or"  (X, Y        : BOOLEAN) return BOOLEAN;
  function "xor" (X, Y        : BOOLEAN) return BOOLEAN;
type INTEGER is {...};
function "="   (LEFT, RIGHT : INTEGER) return BOOLEAN;
function "/="  (LEFT, RIGHT : INTEGER) return BOOLEAN;
```

```
function "<"    (LEFT, RIGHT : INTEGER) return BOOLEAN;
function "<="   (LEFT, RIGHT : INTEGER) return BOOLEAN;
function ">"    (LEFT, RIGHT : INTEGER) return BOOLEAN;
function ">="   (LEFT, RIGHT : INTEGER) return INTEGER;
function "+"    (X : INTEGER) return INTEGER;
function "-"    (X: INTEGER) return INTEGER;
 function "abs" (X: INTEGER) return INTEGER;
 function "+"   (X, Y : INTEGER) return INTEGER;
 function "-"   (X, Y : INTEGER) return INTEGER;
 function "*"   (X, Y : INTEGER) return INTEGER;
 function "/"   (X, Y: INTEGER) return INTEGER;
 function "rem" (X, Y: INTEGER) return INTEGER;
 function "mod" (X, Y: INTEGER) return INTEGER;
 function "**"  (X, Y: INTEGER) return INTEGER;
-- An implementation may provide additional predefined integer types.
 -- It is recommended that the names of such additional types end Integer
 -- as in SHORT ?_INTEGER or LONG_INTEGER
 type FLOAT is digits {...} range {...};
 function "="   (LEFT, RIGHT : FLOAT) return BOOLEAN;
 function "/="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
 function "<"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
 function "<="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
 function ">"   (LEFT, RIGHT : FLOAT) return BOOLEAN;
 function ">="  (LEFT, RIGHT : FLOAT) return BOOLEAN;
 function "+"   (X : FLOAT) return FLOAT;
 function "-"   (X: FLOAT) return FLOAT;
 function "abs" (X: FLOAT) return FLOAT;
function "+"    (X, Y: FLOAT) return FLOAT;
function "-"    (X, Y: FLOAT) return FLOAT;
function "*"    (X, Y: FLOAT) return FLOAT;
function "/"    (X, Y : FLOAT) return FLOAT;
function "**"(LEFT: FLOAT; RIGHT: INTEGER) return FLOAT;
-- An implementation may provide additional
floating_point types.
-- It is recommended that the names of such additional
types end
-- with FLOAT, as in SHORT_FLOAT or LONG ?_FLOAT;
-- The types universal_integer, universal_float, and
universal_fixed
-- are predefined.
function "*" (LEFT : universal_integer; RIGHT :
universal_real) return universal_real;
 function "*" (LEFT : universal_real; RIGHT :
universal_integer) return universal_real;
 function "/" (LEFT : universal_real; RIGHT :
universal_integer) return universal_real;
function "*" (LEFT : any_fixed_type; RIGHT :
```

```
any_fixed_type) return universal_fixed;
function "/" (LEFT : any_fixed_type; RIGHT :
any_fixed_type) return universal_fixed;
-- The following characters comprise the standard ASCII
character set.
-- Character literals corresponding to control
characters are not identifiers;
-- they are underlined in this example.
type CHARACTER is (
        nul,  soh,  sxt,  ext,  eot,  enq,  ack,  bel,
        bs,   ht,   lf,   vt,   ff,   cr,   so,   si,
        dle,  dc1,  dc2,  Dc3,  dc4,  nak,  syn,  etb,
        can,  em,   sub,  esc,  fs,   gs,   rs,   us,
        ' ',  '!',  '"',  '#',  '$',  '%',  '&',  ''',
        '(',  ')',  '*',  '+',  ',',  '-',  '.',  '/',
        '0',  '1',  '2',  '3',  '4',  '5',  '6',  '7',
        '8',  '9',  ':',  ';',  '<',  '=',  '>',  '?',
        '@',  'A',  'B',  'C',  'D',  'E',  'F',  'G',
        'H',  'I',  'J',  'K',  'L',  'M',  'N',  'O',
        'P',  'Q',  'R',  'S',  'T',  'U',  'V',  'W',
        'X',  'Y',  'Z',  '[',  '\',  ']',  '^',  '_',
        ''',  'a',  'b',  'c',  'd',  'e',  'f',  'g',
        'h',  'i',  'j',  'k',  'l',  'm',  'n',  'o',
        'p',  'q',  'r',  's',  't',  'u',  'v',  'w',
        'x',  'y',  'z',  '{',  '|',  '}',        del);

for CHARACTER use (0, 1, 2, ... 126, 127);
package ASCII is

        NUL     : constant CHARACTER := nul;
        SOH     : constant CHARACTER := soh;
        STX     : constant CHARACTER := sxt;
        ETX     : constant CHARACTER := etx;
        EOT     : constant CHARACTER := eot;
        ENQ     : constant CHARACTER := enq;
        ACK     : constant CHARACTER := ack;
        BEL     : constant CHARACTER := bel;
        BS      : constant CHARACTER := bs;
        HT      : constant CHARACTER := ht;
        LF      : constant CHARACTER := lf;
        VT      : constant CHARACTER := vt;
        FF      : constant CHARACTER := ff;
        CR      : constant CHARACTER := cr;
        SO      : constant CHARACTER := so;
        SI      : constant CHARACTER := si;
        DLE     : constant CHARACTER := dle;
```

```
        DC1        : constant CHARACTER := dc1;
        DC2        : constant CHARACTER := dc2;
        DC3        : constant CHARACTER := dc3;
        DC4        : constant CHARACTER := dc4;
        NAK        : constant CHARACTER := nak;
        SYN        : constant CHARACTER := syn;
        ETB        : constant CHARACTER := etb;
        CAN        : constant CHARACTER :=can;
        EM         : constant CHARACTER := em;
        SUB        : constant CHARACTER :=sub;
        ESC        : constant CHARACTER := esc;
        FS         : constant CHARACTER := fs;
        GS         : constant CHARACTER := gs;
        RS         : constant CHARACTER := rs;
        US         : constant CHARACTER := us;
        DEL        : constant CHARACTER := del;

        EXCLAM             : constant CHARACTER := '!';
        SHARP              : constant CHARACTER := '#';
        DOLLAR             : constant CHARACTER := '$';
        QUERY              : constant CHARACTER := '?';
        AT_SIGN            : constant CHARACTER := '@';
        L_BRACKET          : constant CHARACTER := '[';
        BACK_SLASH         : constant CHARACTER := '\';
        R_BRACKET          : constant CHARACTER := ']';
        CIRCUMFLEX         : constant CHARACTER := '^';
        GRAVE              : constant CHARACTER := '`';
        L_BRACE            : constant CHARACTER := '{';
        BAR                : constant CHARACTER := '|';
        R_BRACE            : constant CHARACTER := '}';
        TILDE              : constant CHARACTER := '~';
        QUOTATION          : constant CHARACTER := '"';
        COLON              : constant CHARACTER := ':';
        SEMICOLON          : constant CHARACTER := ';';
        PERCENT            : constant CHARACTER := '%';
        AMPERSAND          : constant CHARACTER := '&';
        UNDERLINE          : constant CHARACTER := '_';
        LC_A               : constant CHARACTER := 'a';
        ...                ...
        LC_Z               : constant CHARACTER := 'z';
  End ASCII;
  -- predefined types and subtypes
subtype NATURAL is INTEGER range 0..INTEGER'LAST;
  subtype POSITIVE  is INTEGER range 1..INTEGER'LAST;
  type      STRING     is array (POSITIVE range <>) of
  CHARACTER;
  function "="    (LEFT, RIGHT : STRING) return BOOLEAN;
```

```
function "/="  (LEFT, RIGHT : STRING) return BOOLEAN;
function "<"   (LEFT, RIGHT : STRING) return BOOLEAN;
function "<="  (LEFT, RIGHT : STRING) return BOOLEAN;
function ">"   (LEFT, RIGHT : STRING) return BOOLEAN;
function ">="  (LEFT, RIGHT : STRING) return BOOLEAN;
function "&"   (LEFT, RIGHT : STRING) return STRING;
function "&"   (LEFT: CHARACTER; RIGHT : STRING)
return STRING;
function "&"   (LEFT: STRING; RIGHT: CHARACTER) return
STRING;
function "&"   (LEFT, RIGHT : CHARACTER) return
STRING;

pragma PACK(STRING);

type DURATION is delta {...} range {...};

-- Predefined exceptions

CONSTARINT_ERROR : exception;
NUMERIC_ERROR      : exception;
PROGRAM_ERROR     : exception;
STORAGE_ERROR      : exception;
T_ERROR      : exception;

End STANDARD;
```

CALENDAR

```
package CALENDAR is
  type TIME is private;

  subtype YEAR_NUMBER is INTEGER range 1901..2099;
  subtype MONTH_NUMBER is INTEGER range  1..12;
  subtype DAY_NUMBER is INTEGER range  1..31;
  subtype DAY_DURATION is INTEGER range  0.0..86_400.0;

  function CLOCK return TIME;
  function YEAR (DATE : TIME) return YEAR_NUMBER;
  function MONTH (DATE : TIME) return MONTH_NUMBER;
  function DAY (DATE : TIME) return DAY_NUMBER;
  function SECONDS (DATE : TIME) return DAY_DURATION;
procedure SLIPT (DATE : in TIME;
            YEAR : out YEAR_NUMBER;
            MONTH: out MONTH _NUMBER;
            DAY: out DAY_NUMBER;
            SECONDS : out DAY_DURATION);
function TIME_OF (YEAR : YEAR_NUMBER;
             MONTH: MONTH _NUMBER;
             DAY  : DAY_NUMBER;
```

```
            SECONDS : DAY_DURATION := 0.0)
            return TIME;

  TIME_ERROR : exception;
  function "+" (X : TIME;Y : DURATION)return TIME;
  function "+" (X : DURATION; Y : TIME)return TIME;
  function "-" (X : TIME; Y : DURATION) return ?TIME;
  function "-" (X : TIME; Y : TIME) return DURATION;
function "<"   (X, Y : TIME) return BOOLEAN;
  function "<="  (X, Y : TIME) return BOOLEAN;
  function ">"   (X, Y : TIME) return BOOLEAN;
  function ">="  (X, Y : TIME) return BOOLEAN;
private
  -- implementation defined
End CALENDAR;
```

IO_EXCEPTIONS

```
package IO_EXCEPTIONS is
  NAME_ERROR    : exception
  USE_ERROR     : exception
  STATUS_ERROR : exception
  MODE_ERROR    : exception
  DEVICE_ERROR  : exception
  END_ERROR     : exception
  DATA_ERROR    : exception
  LAYOUT_ERROR : exception
End IO_EXCEPTIONS;
```

DIRECT_IO

```
with IO_EXCEPTIONS;
generic
type ELEMENT_TYPE is private;
package DIRECT_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, INOUT_FILE, OUT_FILE);
  type COUNT        is range 0..implementation_defined;
  subtype POSITIVE_COUNT is COUNT range 1..COUNT'LAST;

  procedure CREATE (FILE    : in out FILE_TYPE;
                              MODE : in      FILE_MODE := INOUT_FILE;
NAME : in      STRING :=" ";
                            FORM : in      STRING :=" ");  procedure OPEN
(FILE    : in out FILE_TYPE;
MODE  : in FILE_MODE; NAME : in STRING; FORM : in        STRING
:=" ");
  procedure CLOSE   (FILE    : in out FILE_TYPE);
  procedure DELETE (FILE    : in out FILE_TYPE);
```

```
procedure RESET   (FILE   : in out FILE_TYPE;
MODE : in FILE_MODE);
procedure RESET   (FILE   : in out FILE_TYPE);

function MODE    (FILE : in FILE_TYPE) return
FILE_MODE;
function NAME    (FILE : in FILE_TYPE) return STRING;
function FORM    (FILE : in FILE_TYPE) return STRING;
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

procedure READ (FILE : in FILE_TYPE;
ITEM : out ELEMENT_TYPE);
procedure READ (FILE : in FILE_TYPE;
ITEM   : out ELEMENT_TYPE
FORM : in POSITIVE_COUNT);
procedure WRITE (FILE : in FILE_TYPE;
ITEM : in  ELEMENT_TYPE);
procedure WRITE (FILE  : in FILE_TYPE;
ITEM : in ELEMENT_TYPE);
TO: in POSITIVE_COUNT);

procedure SET_INDEX (FILE  : in FILE_TYPE;
TO: in POSITIVE_COUNT);

function INDEX  (FILE : in FILE_TYPE) return
POSITIVE_COUNT;
function SIZE     (FILE : in FILE_TYPE) return COUNT;

function END_OF_FILE (FILE : in FILE_TYPE) return
BOOLEAN;

NAME_ERROR    : exception renames
IO_EXCEPTIONS.NAME_ERROR;
USE_ERROR        : exception renames
IO_EXCEPTIONS.USE_ERROR;
STATUS_ERROR : exception renames
IO_EXCEPTIONS.STATUS_ERROR;
MODE_ERROR: exception renames
IO_EXCEPTIONS.MODE_ERROR;
DEVICE_ERROR   : exception renames
IO_EXCEPTIONS.DEVICE_ERROR;
END_ERROR : exception renames
IO_EXCEPTIONS.END_ERROR;
DATA_ERROR : exception renames
IO_EXCEPTIONS.DATA_ERROR;

private
  -- implementation_defined
End DIRECT_IO;
```

LOW_LEVEL_IO

```
package LOW_LEVEL_IO is
-- declaration of the possible types for DEVICE and DATA
-- declaration of overloaded procedures for these types
procedure SEND_CONTROL (DEVICE : device_type;
DATA : in out data_type);
procedure RECIVE_CONTROL (DEVICE : device_type;

DATA : in out data_type);
End LOW_LEVEL_IO;
```

SEQUENTIAL_IO

```
with IO_EXCEPTIONS;
generic
type ELEMENT_TYPE is private;
package SEQUENTIAL_IO is

type FILE_TYPE is limited private;

type FILE_MODE is (IN_FILE, OUT_FILE);

procedure CREATE (FILE: in out FILE_TYPE;
MODE : in FILE_MODE := OUT_FILE;
NAME : in      STRING :=" ";
FORM : in      STRING :=" ");
procedure OPEN (FILE    : in out FILE_TYPE;
MODE : in      FILE_MODE;
NAME : in      STRING;
FORM : in      STRING :=" ");

procedure CLOSE   (FILE    : in out FILE_TYPE);
procedure DELETE (FILE    : in out FILE_TYPE);
 procedure RESET   (FILE : in out FILE_TYPE;
 MODE : in FILE_MODE);
 procedure RESET   (FILE: in out FILE_TYPE);
 function MODE     (FILE : in FILE_TYPE) return
 FILE_MODE;
 function NAME (FILE : in FILE_TYPE) return STRING;
 function FORM (FILE : in FILE_TYPE) return STRING;
 function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

 procedure READ (FILE : in FILE_TYPE; ITEM : out
 ELEMENT_TYPE);
 procedure WRITE (FILE : in FILE_TYPE;
 ITEM : in  ELEMENT_TYPE);
 function END_OF_FILE (FILE : in FILE_TYPE) return
 BOOLEAN;

 NAME_ERROR    : exception renames
 IO_EXCEPTIONS.NAME_ERROR;
```

```
 USE_ERROR: exception renames  IO_EXCEPTIONS.USE_ERROR;
 STATUS_ERROR : exception renames
 IO_EXCEPTIONS.STATUS_ERROR;
 MODE_ERROR: exception renames
 IO_EXCEPTIONS.MODE_ERROR;
 DEVICE_ERROR: exception renames
 IO_EXCEPTIONS.DEVICE_ERROR;
 END_ERROR: exception renames  IO_EXCEPTIONS.END_ERROR;
 DATA_ERROR: exception renames
 IO_EXCEPTIONS.DATA_ERROR;
 private
 -- implementation_defined
End SEQUENTIAL_IO;
```

SYSTEM

```
package SYSTEM is
 type ADDRESS is implementation_defined;
 type NAME       is implementation_defined;
 SYSTEM_NAME   : constant NAME :=
implementation_defined;
 STORAGE_UNIT : constant:= implementation_defined;
 MEMORY_SIZE  : constant:= implementation_defined;
 MIN_INT: constant:= implementation_defined;
 MAX_INT: constant:= implementation_defined;
 MAX_DIGITS  : constant:= implementation_defined;
 MAX_MANTISSA : constant:= implementation_defined;
 FIND_DELTA: constant:= implementation_defined;
 TICK: constant:= implementation_defined;
subtype PRO?IORITY is INTEGER range {...};
 ...
End SYSTEM;
```

TEXT_IO

```
with IO_EXCEPTIONS;
package TEXT_IO is

  type FILE_TYPE is limited private;

  type FILE_MODE is (IN_FILE, OUT_FILE);
  subtype FIELD is INTEGER range
   0..implementation_defined;
  subtype NUMBER_BASE is INTEGER range 2..16;
  type COUNT is range 0..implementation_defined;
  subtype POSITIVE_COUNT is COUNT range 1..COUNT 'LAST;
  type TYPE_SET is (LOWER_CASE, UPPER_CASE);

  UNBOUNDED : constant COUNT := 0;

procedure CREATE (FILE    : in out FILE_TYPE;
MODE : in       FILE_MODE := OUT_FILE;
```

```
NAME : in      STRING :=" ";
FORM : in      STRING :=" ");
procedure OPEN (FILE    : in out FILE_TYPE;
MODE : in      FILE_MODE;
NAME : in      STRING;
FORM : in      STRING :=" ");

procedure CLOSE   (FILE    : in out FILE_TYPE);
procedure DELETE (FILE    : in out FILE_TYPE);
procedure RESET   (FILE    : in out FILE_TYPE;
MODE : in       FILE_MODE);
procedure RESET   (FILE    : in out FILE_TYPE);

function MODE     (FILE : in FILE_TYPE) return
FILE_MODE;
function NAME     (FILE : in FILE_TYPE) return STRING;
function FORM     (FILE : in FILE_TYPE) return STRING;
function IS_OPEN (FILE : in FILE_TYPE) return BOOLEAN;

procedure SET_INPUT   (FILE    : in FILE_TYPE);
procedure SET_OUTPUT (FILE    : in FILE_TYPE);

function STANDARD_INPUT     return FILE_TYPE;
function STANDARD_OUTPUT  return FILE_TYPE;
function CURRENT_INPUT       return FILE_TYPE;
function CURRENT_OUTPUT     return FILE_TYPE;

procedure SET_LINE_LENGTH (FILE    : in FILE_TYPE;
TO  : in COUNT);
procedure SET_LINE_LENGTH (TO      : in COUNT);

procedure SET_PAGE_LENGTH (FILE    : in FILE_TYPE;

TO  : in COUNT);
procedure SET_PAGE_LENGTH (TO      : in COUNT);

function LINE_LENGTH (FILE    : in FILE_TYPE) return
COUNT;
function LINE_LENGTH return COUNT;

function PAGE_LENGTH (FILE    : in FILE_TYPE) return
COUNT;
function PAGE_LENGTH return COUNT;

procedure NEW_LINE (FILE: in FILE_TYPE;
SPACING : in POSTIVE_COUNT := 1);
procedure NEW_LINE (SPACING : in POSTIVE_COUNT := 1);

procedure SKIP_LINE (FILE : in FILE_TYPE;
SPACING : in POSTIVE_COUNT := 1);
```

```
procedure SKIP_LINE (SPACING : in POSTIVE_COUNT := 1);

function END_OF_LINE (FILE : in FILE_TYPE) return
BOOLEAN;
function END_OF_LINE return BOOLEAN;

procedure NEW_PAGE (FILE : in FILE_TYPE);
procedure NEW_PAGE ;

procedure SKIP_PAGE (FILE : in FILE_TYPE);
procedure SKIP_PAGE ;

function END_OF_PAGE (FILE : in FILE_TYPE) return
BOOLEAN;
function END_OF_PAGE  return BOOLEAN;

function END_OF_FILE (FILE : in FILE_TYPE) return
BOOLEAN;
function END_OF_FILE return BOOLEAN;

procedure SET_COL (FILE : in FILE_TYPE;
TO : in POSTIVE_COUNT);
procedure SET_COL (TO : in POSTIVE_COUNT);

procedure SET_LINE (FILE : in FILE_TYPE;
TO : in POSTIVE_COUNT);
procedure SET_LINE (TO: in POSTIVE_COUNT);

function COL (FILE : in FILE_TYPE) return POSTIVE_COUNT;
function COL return POSTIVE_COUNT;

function LINE (FILE : in FILE_TYPE) return
POSTIVE_COUNT;
function LINE return POSTIVE_COUNT;

function PAGE (FILE : in FILE_TYPE) return
POSTIVE_COUNT;
function PAGE return POSTIVE_COUNT;

procedure GET (FILE  : in FILE_TYPE;
ITEM : out CHARACTER);
procedure GET (ITEM : out CHARACTER);
procedure PUT (FILE  : in FILE_TYPE;
ITEM : in CHARACTER);
procedure PUT (ITEM : in CHARACTER);

procedure GET (FILE  : in FILE_TYPE;
                    ITEM : out STRING);
procedure GET (ITEM : out STRING);
procedure PUT (FILE  : in FILE_TYPE;
```

```
ITEM : in STRING);
procedure PUT (ITEM : in STRING);

procedure GET_LINE (FILE  : in FILE_TYPE;
ITEM : out STRING; LAST : out NATRURAL);
procedure GET_LINE (ITEM : out STRING;
LAST : out NATRURAL);
procedure PUT_LINE (FILE  : in FILE_TYPE;
ITEM : in STRING);
procedure PUT_LINE (ITEM : in STRING);

generic
type NUM is range <>;
package INTEGER_IO is
DEFAULT_WIDTH : FIELD := NUM'WIDTH;
DEFAULT_BASE    : NUMBER_BASE := 10;
procedure GET (FILE  : in FILE_TYPE;
ITEM   : out NUM;
WIDTH : in FIELD := 0);
procedure GET ( ITEM   : out NUM;
WIDTH : in FIELD := 0);
procedure PUT (FILE     : in FILE_TYPE;
ITEM   : in NUM;
WIDTH : in FIELD := DEFAULT_WIDTH;
BASE    : in NUMBER_BASE := DEFAULT_BASE);
procedure PUT ( ITEM   : in NUM;
WIDTH : in FIELD := DEFAULT_WIDTH;
BASE    : in NUMBER_BASE := DEFAULT_BASE);
procedure GET (FROM  : in STRING;
ITEM : out NUM;
LAST : out POSITIVE);
procedure PUT (TO       : out STRING;
ITEM    : in NUM;
BASE    : in NUMBER_BASE := DEFAULT_BASE);
End INTEGER_IO;

generic
type NUM is digits <>;
package FLOAT_IO is
DEFAULT_FORE : FIELD := 2;
DEFAULT_AFT : FIELD := NUM'DIGITS - 1;
DEFAULT_EXP : FIELD := 3;
procedure GET (FILE    : in FILE_TYPE;
ITEM   : out NUM;
WIDTH : in FIELD := 0);
procedure GET ( ITEM   : out NUM;
WIDTH : in FIELD := 0);
procedure PUT (FILE     : in FILE_TYPE;
ITEM   : in NUM;
FORE : in FIELD := DEFAULT_FORE;
```

```
AFT : in FIELD := DEFAULT_AFT;
EXP : in FIELD := DEFAULT_EXP);
procedure PUT ( ITEM   : in NUM;
FORE : in FIELD := DEFAULT_FORE;
AFT : in FIELD := DEFAULT_AFT;
EXP : in FIELD := DEFAULT_EXP);
procedure GET (FROM  : in STRING;
ITEM : out NUM;
LAST : out POSITIVE);
procedure PUT (TO  : out STRING;
ITEM    : in NUM;
AFT : in FIELD := DEFAULT_AFT;
EXP : in FIELD := DEFAULT_EXP);
End FLOAT_IO;

generic
type NUM is delta <>;
package FIXED_IO is
DEFAULT_FORE : FIELD := NUM'FORE;
DEFAULT_AFT : FIELD := NUM'AFT;
DEFAULT_EXP : FIELD := 0;
  procedure GET (FILE    : in FILE_TYPE;
                 ITEM   : out NUM;
                 WIDTH : in FIELD := 0);
  procedure GET ( ITEM   : out NUM;
                 WIDTH : in FIELD := 0);
  procedure PUT (FILE     : in FILE_TYPE;
                 ITEM   : in NUM;
FORE : in FIELD := DEFAULT_FORE;
AFT : in FIELD := DEFAULT_AFT;
EXP : in FIELD := DEFAULT_EXP);
procedure PUT ( ITEM   : in NUM;
FORE : in FIELD := DEFAULT_FORE;
AFT : in FIELD := DEFAULT_AFT;
EXP : in FIELD := DEFAULT_EXP);
procedure GET (FROM  : in STRING;
ITEM : out NUM;
LAST : out POSITIVE);
procedure PUT (TO: out STRING;
ITEM    : in NUM;
AFT : in FIELD := DEFAULT_AFT;
EXP : in FIELD := DEFAULT_EXP);
End FIXED_IO;

generic
type ENUM is (<>);
 package ENUMERATION_IO is
   DEFAULT_WIDTH : FIELD := 0;
   DEFAULT_SETTING : TYPE_SET := UPPER_CASE;
 procedure GET (FILE    : in FILE_TYPE;
```

```
                    ITEM   : out ENUM);
  procedure GET ( ITEM   : out ENUM);
  procedure PUT (FILE    : in FILE_TYPE;
                    ITEM   : in ENUM;
WIDTH : in FIELD := DEFAULT_WIDTH;
SET : in TYPE_SET := DEFAULT_SETTING);
procedure PUT ( ITEM   : in ENUM;
WIDTH : in FIELD := DEFAULT_WIDTH;
SET : in TYPE_SET := DEFAULT_SETTING);
procedure GET (FROM  : in STRING;
ITEM : out ENUM; LAST : out POSITIVE);
procedure PUT (TO  : out STRING;
ITEM    : in ENUM;
SET : in TYPE_SET := DEFAULT_SETTING);
End ENUMERATION_IO;
NAME_ERROR    : exception renames
IO_EXCEPTIONS.NAME_ERROR;
                USE_ERROR          :     exception     renames
IO_EXCEPTIONS.USE_ERROR;
                STATUS_ERROR       :     exception     renames
IO_EXCEPTIONS.STATUS_ERROR;
                MODE_ERROR         :     exception     renames
IO_EXCEPTIONS.MODE_ERROR;
                DEVICE_ERROR       :     exception     renames
IO_EXCEPTIONS.DEVICE_ERROR;
                END_ERROR          :     exception     renames
IO_EXCEPTIONS.END_ERROR;
                DATA_ERROR         :     exception     renames
IO_EXCEPTIONS.DATA_ERROR;
                LAYOUT_ERROR       :     exception     renames
IO_EXCEPTIONS.LAYOUT_ERROR;

 private
  -- implementation_defined
End TEXT_IO;
```

UNCHECKED_CONVERSION

```
generic
  type SOURCE is limited private
  type TARGET is limited private
function   UNCHECKED_CONVERSION   (S   :   in   SOURCE)   return
TARGET;
```

UNCHECKED_CONVERSION

```
generic
  type OBJECT is limited private
  type NAME is access OBJECT
procedure UNCHECKED_DEALLOCATION (X : in out NAME );
```



دليل أسلوب ADA
ADA Style Guide



يمثل أسلوب البرمجة موضوعاً قابلاً للنقاش بشكلٍ حماسيٍ، ولكن هذا يفهم مع ذلك، بشكل سيء. بالرغم من ذلك، إن الأسلوب الذي يُطوّر به نظام، سيؤثر كثيراً على قابلية الفهم، وعلى الصيانة، وعلى أداء ذلك النظام. طوال أمثلتنا و مناقشاتنا، حاولنا إيضاح أسلوب برمجة يلائم هذه الأهداف، وبنفس الوقت، يستثمر قوة لغة ADA. في هذا الملحق، نعرض المستويات الثلاثة من أسلوب البرمجة، ملخصين بذلك كل ما اقترحناه طوال الكتاب.

فعندما نتكلم عن أسلوب البرمجة، فإن معظم الناس يفكرون بالمعايير مثل«يجب ألا تستخدم Goto!»، أو «كل وحدة يجب أن تنطبق على صفحة واحدة!». وإن هكذا قواعد ليست فقط صنعية، ولكنها أيضاً تجبر المبرمج على الإنشغال بمسائل قليلة الفعالية، كنقيضٍ لفلسفة التصميم العام المحتاجة في أي نظام. ومن جهة أخرى، إن إتباع أسلوبٍ برمجي يبلور البنية العامة لنظام بطريقةٍ تعكس مباشرةً رؤية العالم الحقيقي يكون أفضل طريقة.

وهكذا أسلوب، يمكن فقط أن يوصف على شكل أدلة – بما أنّ ADA لغة ضخمة، صُممت من أجل حل مسائل معقدة، فمن غير الممكن أن تصف قاعدة من أجل كل تطبيق ممكن. وأكثر من ذلك، فإضافة قواعد أسلوب اللغة (تُجبر المبرمج ليتعلم الكثير من اللغة) غالباً ما تخدع، أو حتى تحرم المبرمج من استخدام خصائص اللغة التي من الواضح أنها ستكون أكثر فاعلية أو قابلية للقراءة.

ومن هذا المنظار، فإننا نقسم مدى الأدلة لأسلوب البرمجة إلى ثلاث مناطق أساسية، تُدعى:

- أسلوب التصميم (Style of design).
- أسلوب تطبيق اللغة (Style of applying the language).
- أسلوب التقديم (Style of presentation).

أسلوب التصميم (Style of design):

سُئِل مرة مجموعة من المبرمجين المحترفين عن كيفية تصميمهم لنظمهم. فكان الجواب الوحيد نظرات الإستغراب. وبعد قليلٍ من الأسئلة الدقيقة، بدى سبب عدم إجابتهم: حيث لا يصمم هؤلاء المبرمجين حلولهم، إنهم فقط يكتبون الترميز ببساطة.

ولخلق حلول موثوقة لمسائل معقدة، يجب أن نصمم لها بُنى منطقية ومتماسكة. وإن البنية الهادفة، هي التي تجعل الحل قابلاً للفهم وللصيانة. وحتى إذا أُستخدِمت أفضل الأدوات لتطوير الحل، فلا يمكن للغات البرمجة الأكثر تعبيراً في العالم أن تحسّن التصميم السيء.

ومثلما ذكرنا في كثيرٍ من الأحيان، إننا نفكر في مسائل من عالمٍ حقيقي، بدلالة أسماء وأفعال. ولغة ADA، تتبع حلولنا من العالم الحقيقي، وذلك بتوفير وسيلةٍ تصف بوضوح الأغراض، والعمليات بطريقةٍ متوازنةٍ. ولدعم هذه الرؤية من العالم، استخدمنا طريقة التصميم غرضية التوجه في هذا الكتاب (لاحظ الفصل 5). ويبدو أنّ هذا الأسلوب يعمل بشكل حسن لمسائل ذات مجالاتٍ واسعة. وأكثر من ذلك، لقد دعمت ADA هذا الأسلوب بشكلٍ حسن.

ويتمثل جوهر طريقة التصميم هذه بإمكانية تجريد المستخدم للمعطيات، والخوارزميات في مستوى معطى، مخفياً في نفس الوقت التفاصيل غير الضرورية، لذلك المستوى من التجريد (معيار تحليل Parnas). وعلى عكس تقنيات التصميم الوظيفية البحتة، تُدرك هذه الطريقة أهمية الأغراض و العمليات في الحلول. وبمعناه الأوسع، تقدم أداةً لإدارة تعقيد الحلول.

ونعرض في الأدلة التالية أسلوب التصميم:

- استخدم أسلوباً غرضي التوجه لتجريد بنية المعطيات، والتحكم.
- مهما يكن مستوى التجريد، تذكر عدد الكيانات التي يمكن للمبرمج أن يُديرها في وقت واحد.
- يجب أن تكون الوحدات البرمجية ذات تماسك قوي، وارتباط ضعيف (لاحظ الفصل 4).

ويمكن أن تقول لنفسك، بأنّ تطبيق هكذا أسلوب غير فعّال، خصوصاً في نظم الزمن الحقيقي، بسبب الكلفة الزمنية الزائدة الناتجة عن عدد الوحدات. ولتقليص هذه الكلفة الزائدة، يمكننا دائماً استخدام العملياتي Inline، الذي يحذف تمرير المعاملات و الارتباطات الموافقة، ويحفظ في الوقت ذاته وضوحية اللغة العالية المستوى (لاحظ الفصل 20).

وأكثر من ذلك، يكون النظام من حيث الأصل أكثر وثوقية، إذا عكس مباشرة العالم الحقيقي –وبالفعل إن أجزاء الحل التي لا توافق العالم الحقيقي، هي التي تكون أكثر عرضة لتصرفات منحرفة. ما عدا ذلك، يتم تطبيق القاعدة 10-90، وهذا يعني أنّ 90% من موارد النظام تُستهلك بـ 10% من الترميز. فإذا وجدت عقدة اختناق للفاعلية، فمن غير المجدي تخريب بنية الحل الكلية من أجل حل المسألة. وأفضل طريقة تتمثل بالتحقق من هذه الـ 10%، واستخدام البُنى من أجل تحسين الأداء محلياً (مثل إعادة تصميم ذلك المقطع من الترميز، أو استخدام توصيفات تمثيل ADA). ومن منظور دورة حياة النظم، فإن تصميم نظام بأسلوب جيد سيحقق مردوداً عالياً. ومن جهة أخرى، لن تغتفر بنية تصميم ضعيفة.

أسلوب تطبيق اللغة (Style of applying the language):

لا يمكن لأحد أن يدعي معرفة كل شيءٍ معروفٍ باللغة العربية، ولكننا نتدبر أمورنا بما نعرفه منها. وبشكل عام، نستخدم فقط مجموعة بسيطة من اللغة العربية، ولكن في بعض الأحيان نلجأ لبُنى أكثر تعقيداً (مثل عدم التباس مرجعي) عندما نحتاج للتعبير عن طيفٍ أكثر دقة من المعاني. ويطبق نفس المفهوم في لغات البرمجة. معظم الوقت، نطبق فقط مجموعة جزئية من بُنى اللغة؛ وفي بعض الأحيان نطبق بُنى أكثر تعقيداً لأنها تحل مسائلنا بفاعلية أكثر، أو بوضوح أكثر. وعندما نتكلم عن أسلوب تطبيق أدواتنا، فإننا نعني بذلك أنه يجب أن نشجع استخدام بعض بُنى اللغة، بينما نثبط (دون أن نبعدها بالضرورة) استخدام البقية.

ففي أمثلتنا، لاحظنا بُنى لغة تكون في بعض الأحيان خطرة، أو ليست واضحة. ومهما كانت تسهيلات اللغة، فإنه توجد طريقة جيدة، وطريقة سيئة لاستخدام أي بنية. وبما أنّ الأدلة ذات حجم تصعب جدولتها هنا، فسنقدم بدلاً من ذلك أدلة متغيرة. وبشكل خاص، نقترح التطبيقات التالية من أجل تسهيلات ADA:

- ***البرامج الجزئية:***

وحدات البرنامج الأساسية.

تعريف التحكم الوظيفي.

تعريف العمليات على الأنواع.

- ***الحزم البرمجية:***

مجموعات مسماة من التصريحات.

تجميع وحدات برنامج مرتبطة.

أنواع معطيات مجردة.

آلات حالات ـ مجردة.

- ***المهام***

الأعمال المتوازية.

تدوير الرسائل.

التحكم بالموارد.

المقاطعات.

- ***وحدات برامج مولدة:***

المكونات البرمجية التي يمكن إعادة استخدامها.

التحكم بالرؤية.

- ***الإستثناءات:***

اكتشاف شروط الأخطاء و تجنبها.

اكتشاف و تصحيح شروط متوقعة لكنها استثنائية.

ويجب أيضاً، أن يكون المبرمج منتبهاً لكيفية تسميته لكيانات اللغة. وحتى بخصائص بسيطة للغة، مثل الرمز "_" داخل المعرّفات، أو استخدام ارتباط معاملات مسماة، يمكن إنتاج ترميز مقروء بشكل ممتاز. وتقدم الأدلة التالية لتسمية كيانات برنامج:

- يجب تسمية البرامج الجزئية بجمل فعلية ؛ التوابع الفرعية التي تعيد قيماً منطقية ويجب تسميتها مع جمل تستخدم فعل الكون.

Start_Mixing_Process, Sort_List, Is_Not_Empty

- ويجب تسمية الحزم البرمجية بجمل اسمية.

Math_Functions, Earth_Constants

- ويجب تسمية المهام بجمل اسمية (عادة تشير لعمل).

Timer, Message_Router, List_Searcher

ويجب تسمية الأنواع كجمل اسمية مشتركة.

Tree, Linked_List, Index

- ويجب تسمية الأغراض كجمل اسمية خاصة.

My_Tree, Personnel_Linked_List, Data_Base_Index

وكيفما كانت طريقة استخدامنا للغة، فإننا نقترح الدليل الأساسي التالي: إذا وجد خيار بين بُنيتين أو أكثر، فاختر البنية الأكثر وضوحاً.

أسلوب التقديم (Style of presentation):

يجب إعارة الاهتمام الأكثر لهذا الجزء، لأنه الأسهل للوصف، ويؤكد على التنفيذ الآلي. وكنقيض لأسلوب تطبيق اللغة، الذي يتطلب استخدام بعض البُنى، يعود أسلوب التمثيل إلى التصميم الفيزيائي لترميز المنبع. وتتمثل الفائدة من مستوى الأسلوب هذا بجودة القراءة الإنسانية، ولا يوجد أي تأثير على بيئة التنفيذ.

وتتضمن معظم الأدلة المرتبطة بالتمثيل اقتراحاتٍ من أجل ترتيب النص، واستخدام التعليقات، والتنسيق النصي. وطوال هذا الكتاب، تمت كتابة الترميز

باستخدام طريقة ترتيب نص مطلوبة. وبخصوص أسلوب التعليق، فإن ترميز ADA ضخم وبجزءٍ كبيرٍ منه هو التوثيق الذاتي (على الأقل على المستوى المحلي) بفضل استخدام هكذا خصائص، مثل ارتباط معاملاتٍ مسمّاة. وأخيراً، إذا تمّ استثمار وسيلة الترجمة المنفصلة في اللغة، (وهذا ما يجب عمله)، فسيكون تقديم النص مهماً، فقط في مستوى منخفض. ولا يوجد بالضرورة تنظيم عام لنص على شكلٍ خطي في معظم اللغات. فعلى العكس اللغة تدعم طبولوجيا متعددة الأبعاد.

وبإعطاء هذه الفلسفة، يمكن طرح أدلة التقديم، كمايلي:

- اتبع أسلوب ترتيب النص المطلوب.
- استخدم الفراغات بمهارة من أجل تحسين وضوحية النص.
- اجمع كل الكيانات المرتبطة.
- يمكن تحسين قابلية القراءة باستخدام تقنياتٍ بسيطةٍ، مثل ترتيب النقطتين العموديتين، ترتيب لائحة من التصريحات أبجدياً.

وفي الدليل الأخير، لاحظ بأنّه حتى البرنامج يجب أن يملك إغراء جمالياً. ويجب أن يبدو البرنامج مريحاً للعين، ومفهوماً عند فحصه عن قرب. وأخيراً، تذكر الحقيقة أنّه يمكن كتابة الترميز مرة واحدة، لكن تتم قراءته كثيراً. وعند القيام بالبرمجة فكر بالقارئ، وليس بالكاتب.

المصطلحات
Glossary

لقد تم عمل الجدول التالي بسماحٍ من وزارة الدفاع الأميركية، ويحوي على عناصر إضافية، تم الإشارة إليها بنجمة. وقد رتب حسب تسلسل اللغة الإنكليزية:

Abstraction التجريد: هو رؤيتنا لمجمل فضاء المسألة. وفي الواقع، إن كل ما يمكن معرفته هو تجريد. وكل تجريد، هو جزء من سلم تجريد حيث يزرع كل مستوى منه في مستوى أدنى.

Accept Statement تعليمة القبول: انظر إلى المدخل .Entry

Access Type نوع الوصول: إن قيمة نوع الوصول، إما معدومة، أو قيمة تدل على غرض خلق من مخصص. والغرض المعيّن يمكن أن يقرأ، أو أن يرسل عبر قيمة الوصول. وتحديد نوع الوصول يعين نوع الأغراض المحددة بقيم نوع الوصول. (انظر أيضاً Collection).

Actual Parameter المعامل الحالي: (انظر Parameter).

Aggregate تراكم: يعطي قيمة النوع المركب. وتحدد القيمة بإعطاء قيمة كل مركب. يمكن أن يستخدم الإسناد بالاسم لتحديد أي قيمة هي مسندة لأي مركب.

Allocator المخصص: إن تقييم المخصص يخلق غرضاً ويعيد قيمة وصول جديدة هي التي تحدد الغرض.

Array Type نوع المصفوفة: إن قيمة نوع المصفوفة، مؤلفة من مركبات هي عادة أنواع أو أنواع جزئية مختلفة. ومن أجل كل مركب من قيمة التسجيلة أو غرض، فإن تحديد نوع التسجيلة يعين معرفاً Identifier يحدد بطريقة واحدة مركب التسجيلة.

Assignment الإسناد: هو العملية التي تحل فيها قيمة جديدة محل قيمة سابقة لمتحول. وتعليمة الإسناد تحدد متحولاً على يسارها أما على اليمين فيكون هنالك تعبير يحدد قيمة جديدة للمتحول.

Attribute الواصف: إن تقييم الواصف يعطي مواصفةً مسبقةً لكيان مسمى. وبعض هذه الواصفات توابع.

Block Statement تعليمة الكتلة: وهي تحتوي على سلسلة تعليمات، ويمكنها أن تحوي أيضاً جزءاً تصريحياً، ومعالجات استثناء، تأثيراتها موضعية.

Body الجسم: ويحدد تنفيذ البرنامج الجزئي، وحزمة برمجية أو مهمة. وتعتبر طبقة الجسم شكلاً من أشكال الجسم الذي يشير إلى أنّ هذا التنفيذ محدد بوحدة جزئية منفذة بشكل منفصل.

Character* محرف: هو كل رمز من رموز ASCII، المستخدمة في كتابة البرامج أو المعطيات. المحارف البيانية لها تمثيل مقروء، ومحارف التحكم تملك واصفات مرئية تتعلق بالترجمة.

Collection مجموعة: هي مجموعة الأغراض المخلوقة بواسطة تقييم المخصصات من أجل نوع الوصول.

Compatible* متوافق: من وجهة نظر القيود يكون تصريح ما متوافقاً، إذا كان قيدها يحقق غاية الأصل أو الأساس.

Compilation Unit وحدة الترجمة: وتمثل تصريحاً أو جسم وحدةٍ برمجية، الممثلة بالترجمة كنص مستقل. ويمكن أن تُسبق عبارة السياق، المسمية لوحدات برمجية أُخرى، المتعلقة بسياق أو أكثر لـ "with".

Component عنصر مكوّن: يمثل العنصر قيمة صغرى من قيمة أكبر، أو غرض جزئي من غرض أكبر.

Composite Type النوع المركب: إن النوع المركب، هو نوع تُمثل قيمه بعناصر، وهنالك نوعان من الأنواع المركبة: أنواع المصفوفة وأنواع التسجيلة.

Constant الثابت: (انظر الغرض Object).

Constraint القيد: يعني مجموعةً جزئيةً لقيم النوع، وإن قيمه في تلك المجموعة الجزئية تحقق القيد.

Context Clause عبارة السياق: (انظر وحدة الترجمة Compilation Unit).

Conversion التحويل: يمثل عملية تحويل نوع إلى آخر.

Declaration التصريح: تصريح ما يربط معرّفاً (أو أي شيء آخر) بكيان ما. هذا الربط يكون فعالاً في منطقة من النص تسمى مدى التصريح (Scope). في مدى التصريح يوجد أماكن يمكن فيها استعمال القيود للرجوع إلى الكيان المصرح عنه المرتبط. في هكذا أمكنة يكون المحدد اسماً بسيطاً للكيان. والاسم يدل على الكيان المرتبط.

Declarative Part قسم التصريح: جزء التصريح، وهو سلسلة تصريحات يمكن أن تحتوي أيضاً معلوماتٍ قريبة مثل أجسام البرنامج الجزئي وسياقات تمثيل.

Denote: انظر .Declaration

Derived Type النوع المشتق: إن النوع المشتق، هو نوعٌ عملياته وقيمة مطابقة لعمليات وقيم النوع الموجود. والنوع الموجود يسمى النوع الأب للنوع المشتق.

Designate* انظر .access type , task

Direct Visibility الرؤية المباشرة: انظر .Visibility

Disambiguation* منع الالتباس: هي عملية الاختيار لكيانٍ مسمّى من بين عدة أسماء ذات تحميل زائد.

Discrete Type النوع المتقطع: إن النوع المتقطع، هو نوع له جملة منظمة من القيم المميزة.

والأنواع المتقطعة، هي الأنواع المرقمة والأنواع الصحيحة. وتستخدم من أجل الفهرسة والتكرار، ومن أجل الاختيارات في تعليمات الـ "Case"، ومتحولات التسجيلة.

Discriminant المميز: المميز، هو عنصر خاص لغرض أو قيمة نوع تسجيلة. والأنواع الجزئية لعناصر أخرى، أو حتى وجودها، أو غيابها يمكن أن تتعلق بقيمة المميز.

Discriminant Constraint قيد المميز: إن قيد المميز على نوع التسجيلة، أو نوع المشتق يعين قيمة لكل مميز للنوع.

Elaboration إعداد: إعداد تصريح، هو الإجرائية التي ينتج فيها التصريح أثاره (مثل خلق غرض)، والتنفيذ.

Entry المدخل: يستخدم المدخل من أجل التواصل بين المهام. ويستدعى المدخل من الخارج، كما يستدعى أي برنامج جزئي. ويحدد سلوكه الداخلي بواسطة تعليمة أو عدة تعليمات "accept"، والتي تحدد الأعمال الواجب تنفيذها عندما يستدعى المدخل.

Enumeration Type النوع المرقم: هو نوع متقطع قيمه ممثلة بحروف رقمية معطاة بوضوح في تصريح النوع. وهذه الحروف الرقمية هي معرفات، أو حروف معارف.

Error الخطأ: هو شرط يشذ عن القاعدة، أو هو شرط منطقي. يمكن تصنيف الأخطاء في ثلاث فئات: تلك التي يجب أن تكتشف عند الترجمة (الشذوذ عن قاعدة اللغة)، وتلك التي يجب أن تكتشف عند التنفيذ (بإبراز استثناء)، وتلك التي هي شذوذ عن قواعد اللغة، والتي يجب أن يحترمها كل برنامج في لغة ADA، والتي ليس من الضروري أن تكون مدققة بمترجم ADA. نعتبر أن كل برنامج يشذ عن تلك القاعدة، هو برنامجاً خاطئاً، وأثر البرنامج غير متوقع.

Evaluation التقييم: إن تقييم تعبير ما هو عملية تحسب بها قيمة التعبير. وهذه العملية تتم عند تنفيذ البرنامج.

Exception الاستثناء: إن الاستثناء هو حالة خطأ، يمكن أن تحصل عند تنفيذ البرنامج. إبراز استثناء هو ترك التنفيذ الطبيعي للبرنامج بحيث يشار إلى أن خطأ ما قد حدث. وإن معالجة الاستثناء هي جزء من نص البرنامج المحدد لجواب الاستثناء. وإن تنفيذ هذا النص يدعى معالجة الاستثناء.

Expanded Name الاسم الموسع: يصرح الاسم الموسع عن كيان مصرح به مباشرة في البناء. والاسم الموسع له شكل مركب مختار: السابقة تصرح عن البناء (وحدة برامج، تعليمة ("Block"، "Loop"، أو"accept")، والمختار هو الاسم البسيط للكيان.

Expression التعبير: إن التعبير يحدد حساب القيمة.

Fixed-point Type نوع النقطة الثابتة: انظر النوع الحقيقي .Real Type

Floating- point Type نوع النقطة العائمة: انظر النوع الحقيقي .Real Type

Formal parameter المعامل الصوري: انظر المعامل .Parameter

Function التابع: انظر البرنامج الجزئي.Subprogram

Generic Unit الوحدات المولدة: الوحدة المولدة، هي نموذج لمجموعة حزم برمجية، والبرنامج الجزئي، أو الحزمة البرمجية المخلوقة باستخدام هذا النموذج، يسمى نسخ جزئية أو الوحدة المولدة. إن النسخ المولد هو شكل من التصريح يخلق النسخة. والوحدة المولدة تكون مكتوبة مثل برنامج جزئي، أو حزمة برمجية، ولكن مع توصيف مسبوق بجزء صوري مولد يمكنه أن يولد معاملات صورية مولدة. والمعامل الصوري المولد هو نوع، أو برنامج جزئي، أو حتى غرض. وإن الوحدة المولدة هي واحدة من أشكال الوحدة البرمجية.

Handler معالج: انظر .Exception

***Identifier المعرف**: هو أحد عناصر مفردات الأساس للغة، ونستخدم معرفاً كاسم كيان، أو كلمة محجوزة.

Index فهرس: انظر النوع المصفوفة .Array Type

Index Constraint قيد الفهرسة: إن قيد الفهرسة لمصفوفة يحدد الحدود الدنيا والعليا لكل مجال فهرسة لنوع مصفوفة.

Indexed Component عنصر الفهرسة: يصرح عنصراً من مصفوفة. وهو شكل من اسم يحتوي على تعابير تحدد قيم قرائن وعنصر المصفوفة. عنصر الفهرسة يمكنه أيضاً التصريح عن مدخل في عائلة مداخل.

Instance النسخة: انظر الوحدة المولدة .Generic Unit

Integer Type النوع الصحيح: هو نوع متقطع، تمثل قيمه جميع الأرقام الصحيحة لمجال محدد.

Lexical element عنصر مفردات: هو معرف، أو حرف، أو محدد أو تعليق.

***Library Unit وحدة المكتبة**: وحدة الترجمة التي ليست وحدة جزئية لوحدة أخرى. ووحدات المكتبة، هي جزء من مكتبة البرنامج.

Limited Type النوع المحدود: هو نوع من أجله لا الإسناد، ولا المقارنة المحدودة مسبقاً لمساواة مصرح بها ضمنياً. وكل أنواع المهام هي محدودة. والنوع الخاص يمكن أن يعرف كنوع محدود. والمساواة يمكن أن يصرح بها بوضوح كنوع محدود.

Literal حرفي: يمثل الحرف قيمة حرفية، أي بواسطة أحرف، ومحارف أخرى. الحرف هو حرف عددي، أو حرف رقمي، أو حرف محرف، أو حرف سلسلة.

Mode طريقة: انظر المعامل .Parameter

Model Number رقم النموذج: إن رقم النموذج، هو قيمة نوع حقيقي يمكن أن تكون ممثلة تماماً. والعمليات لنوع حقيقي محددة بحدود العمليات على أعداد النماذج للنوع. وإن خواص الأعداد النماذج وعملياتها، هي خواص أصغرية محفوظة بكل زروع النوع الحقيقي.

Name الاسم: الاسم، هو تعليمة تمثل كياناً. ونقول أن الاسم يصرح عن الكيان، وأن الكيان هو مدلول الاسم. انظر أيضاً .Prefix, Declaration

Named Associationالارتباط الاسمي: إن الارتباط الاسمي يحدد ارتباط عنصر إلى عدة مواضع في لائحة عن طريق تسمية المواضع.

Number* العدد: هو حرف من نوع صحيح أو حقيقي .Universal

إن الرقم المسمى، هو عدد ثابت يمكن أن يرجع إليه المحدد المعطى في تصريح الرقم.

Object الغرض: يحتوي الغرض على قيمة. وإن برنامجاً ما يخلق غرضاً عن طريق إعداد تصريح الغرض، أو عن طريق تقييم مخصص. والتصريح، أو المخصص يحدد نوعاً من أجل الغرض. والغرض لا يمكن أن يحتوي إلا على قيم لهذا النوع.

Operation العملية: إن العملية، هي فعل عنصري مرتبط بنوع، أو عدة أنواع. وقد تكون ضمنياً مصرحة بواسطة تصريح النوع، أو هي برنامج جزئي له معامل، أو نتيجة لهذا النوع.

Operator المؤثر: والمؤثر، هو عملية لها واحد، أو عدة متأثرات (Operands). والمؤثر الأحادي يكتب أمام المتأثر. والمؤثر الثنائي يكتب بين متأثرين. وهذا الترميز (notation)، هو شكل خاص من استدعاء التابع. ويمكن أن يتم التصريح عن المؤثر، كما هي الحال في التابع. وإن كثيراً من المؤثرات مصرح بها ضمنياً بواسطة التصريح عن نوع (مثال، إن أغلب تصريحات النوع تتضمن تصريح المؤثر "مساواة" من أجل قيم هذا النوع).

Overloading التحميل الزائد: يمكن أن يكون للمعرف عدة دلائل مختلفة في مكان ما من نص البرنامج. هذه الخاصة تسمى زيادة تحميل. مثال: إن حرف المرقم ذي التحميل الزائد يمكن أن يكون معرفاً يظهر في تعاريف نوعين أو عدة أنواع رقمية. والمدلول الفعال لمعّرفٍ ذي تحميلٍ زائد، يكون محدداً بالنص الكامل. البرامج الجزئية، والتراكمات، المخصصات، والحروف السلسلة يمكن أيضاً أن تكون ذات تحميل زائد.

Package الحزمة البرمجية: تعين مجموعة كيانات مرتبطة منطقياً، مثل الأنواع، والأغراض التي تملك هذه الأنواع، وبرامج جزئية مع معاملاتٍ تملك هذه الأنواع. والحزمة البرمجية تكون مكتوبةً على شكل تصريح حزمة برمجية، وعلى شكل جسم حزمة برمجية. والتصريح عن حزمة برمجية له جزء مرئي يحتوي على تصريحات جميع الكيانات التي يمكن أن تستخدم بوضوح خارج الحزمة البرمجية. ويمكن أن يكون لها أيضاً جزء خاص يحتوي على تفاصيل بنيوية تكمل تحديد الكيانات المرئية لكن لا تخص مستخدم الحزمة البرمجية. وإن جسم الحزمة البرمجية يحتوي على زروعات البرامج الجزئية (وكذلك المهام أو الحزم البرمجية الأخرى)، التي تم تحديدها في التصريح عن الحزمة البرمجية. وتعتبر الحزمة البرمجية شكلاً من أشكال وحدة البرنامج.

Parameter المعامل: إن المعامل هو أحد الكيانات المسمّاة، والمرتبطة بالبرنامج الجزئي، أو المدخل، أو الوحدة المولدة، ويستخدم للتواصل مع جسم البرنامج الجزئي الموافق أو تعليمة الـ accept، أو جسم المولد. والمعامل الصوري، هو معرّف يدل على الكيان المسمى داخل الجسم. أما المعامل الفعلي، فهو الكيان

الخاص المرتبط بالمعامل الصوري بواسطة استدعاء البرنامج الجزئي، أو استدعاء المدخل، أو نسخ المولد. وطريقة المعامل الصوري تحدد فيما إذا كان المعامل الفعلي المرتبط يقدم قيمةً للمعامل الصوري، أو إذا كان المعامل الصوري هو الذي يقدم قيمة للمعامل الفعلي أو الاثنين معاً. وإن ارتباط المعاملات الفعلية بالمعاملات الصورية يمكن أن يحدد بالإرتباط الاسمي، أو بالارتباط الموضعي، أو بالاثنين معاً.

Parent Type النوع الأب(الأصل): انظر النوع المشتق .Derived Type

Positional Association الارتباط الموضعي: يحدد الارتباط الموضعي ارتباط عنصرٍ بموضعٍ ما في لائحة ما عن طريق استخدام الموضع ذاته في النص لتحديد العنصر.

Pragma العملي: يقوم العملي بتقديم المعلومات للمترجم.

Prefix السابقة: وتستخدم كجزء أول لبعض أشكال الاسم. والسابقة، هي استدعاء تابع، أو اسم.

Private Part القسم الخاص: انظر .Package

Private Type النوع الخاص: هو نوع تكون بنيته ومجموعة قيمه محددة بوضوح، ولكن غير جاهزة مباشرة لاستخدام النوع. ويعرف النوع الخاص بمميزاته، وبمجموعة العمليات المحددة له. النوع الخاص، والعمليات المطبقة عليه تكون محددة في الجزء المرئي من الحزمة البرمجية، أو في الجزء الصوري المولد. الإسناد، والمساواة، وعدم المساواة تكون أيضاً محددة في الأنواع الخاصة إلا إذا كان النوع الخاص محدوداً.

Procedure الإجرائية: انظر .Subprogram

Program البرنامج: يكوِن البرنامج مؤلفاً من عدد من وحدات الترجمة إحداها برنامج جزئي يسمى البرنامج الرئيسي. يتطلب تنفيذ البرنامج تنفيذ البرنامج الرئيسي الذي يمكن أن يستدعي البرامج الجزئية المصرح عنها في وحدات ترجمة أخرى من البرنامج.

*Program Library مكتبة البرامج: وهي جزء من بيئة البرمجة في APSE) ADA) المتعرف عليها من قبل مترجم لغة ADA، والتي تستخدم لتجميع وحدات البرمجة.

Program Unit وحدة البرنامج: هي وحدة مولدة، أو حزمة برمجية، أو برنامج جزئي، أو وحدة مهمة.

Qualified Expression التعبير الموصف: هو تعبير مسبوق بتعليمة عن النوع الجزئي، أو النوع الجزئي. هكذا توصيف يكون مستخدماً إذا كان التعبير غامضاً أثناء غيابه (مثلاً نتيجة التحميل الزائد).

Raising an Exception إبراز الاستثناء: انظر.Exception

Range المجال: هو مجموعة قيم مستمرة لنوع سلمي. ويحدد المجال بإعطاء الحدود الدنيا، والعليا للقيم. ونقول عن قيمة أنها من المجال إذا ما انتمت إليه.

Range Constraint قيد المجال: هو نوع يعين المجال وعندها يحدد المجموعة الجزئية لقيم النوع التي تنتمي إلى المجال.

Real Type النوع الحقيقي: هو نوع تمثل قيمه تقريب الأعداد الحقيقية. وهناك نوعان من الأنواع الحقيقية: الأنواع ذات النقطة الثابتة، والمحددة بهامش خطأ مطلق، والأنواع ذات النقطة العائمة المحددة بهامش خطأ نسبي معبر عنه كعدد من أرقام عشرية ذات معنى.

Record Type النوع تسجيلة: وتتألف قيمة النوع تسجيلة من عناصر هي عادة أنواع أو أنواع جزئية مختلفة. ومن أجل كل عنصر لقيمة تسجيلة، أو لغرض تسجيلة، فإن تعريف النوع تسجيلة يعين معرفاً يحدد بطريقة وحيدة العنصر في التسجيلة.

Renaming Declaration التصريح عن إعادة التسمية: هو تصريح عن اسم آخر لكيان ما.

Rendezvous الموعد: هو التداخل الذي يحصل بين مهمتين متوازيتين عندما تستدعي مهمة مدخلاً لمهمة أخرى، وعندما تكون التعليمة "accept" الموافقة قد نفذت من قبل المهمة الأخرى على حساب المهمة المستدعية.

Representation Clause عبارة التمثيل: يوجه المترجم سياق التمثيل لاختيار تنقل نوع أو غرض أو مهمة من أجل مواصفات الآلة التحتية التي تنفذ برنامجاً. وتحدد سياقات التمثيل في بعض حالات التنقل تماماً. وفي حالات أخرى لا تقدم سوى معايير لاختيار التنقل.

Satisfy انظر .Constraint Subtype

Scalar Type النوع السلمي: هو غرض، أو قيمة لنوع سلمي لا تملك عناصر. ويكون النوع السلمي إما متقطعاً، أو حقيقياً، وتكون قيم النوع السلمي مرتبة.

Scope المدى: انظر .Declaration

Selected Component العنصر المنتخب(المختار): هو اسم مؤلف من سابقة ومعرف يدعى المختار Selector. وتستخدم العناصر المختارة لترميز عناصر تسجيلة، أو مداخل، أو أغراض معينة بقيم وصول، وتستخدم أيضاً موسعة.

Selector الناخب (المختار): انظر .Selected Component

Semantics* علم الدلالة: مدلول لبنية كيان معطى.

Simple Name الاسم البسيط: انظر .Declaration Name

Statement التعليمة: وهي تحدد فعلاً أو عدة أفعال واجب تنفيذها خلال تنفيذ البرنامج.

Subprogram البرنامج الجزئي: هو إما إجرائية أو تابع. وتحدد الإجرائية سلسلة أفعال تعطيها التعليمة استدعاء إجرائية. أما التابع، فيحدد سلسلة أفعال، ويعيد أيضاً قيمة تسمى النتيجة. إن استدعاء تابع هو تعبير. ويكتب البرنامج على شكل تصريح لبرنامج جزئي الذي يعيين اسمه، ومعاملاته الصورية، ونتيجته (من أجل تابع). ويكتب أيضاً على شكل جسم لبرنامج جزئي الذي يحدد سلسلة الأفعال. وإن استدعاء برنامج جزئي يعين المعاملات الفعلية التي

يجب ربطها بالمعاملات الصورية. والبرنامج الجزئي، هو شكل من أشكال وحدات البرنامج.

Subtype النوع الجزئي: هو نوع يميز مجموعة جزئية من قيم النوع. والمجموعة الجزئية تكون محددة بمحدد على النوع. وكل قيمة في مجموعة القيم لنوع جزئي تنتمي إلى نوع جزئي، وتستجيب للقيد الذي يحدد النوع الجزئي.

Subunit الوحدة الجزئية: انظر .Body

syntax* القواعد اللغوية: هي قواعد اللغة (The Grammar) التي تحدد كيفية تجميع سلسلة لتقديم برامج أصلية صحيحة تماماً.

Task المهمة: تعمل المهمة بالتوازي مع باقي أجزاء البرنامج. وتكتب على شكل توصيف مهمة (والتي تحدد اسم المهمة وأسماء المعاملات الصورية لداخلها)، وعلى شكل جسم المهمة الذي يحدد التنفيذ. أما وحدة المهمة، فهي شكل من أشكال وحدة البرنامج. وأما نوع المهمة، هو نوع يسمح بتصريحات لاحقة لعدد من المهام المماثلة للنوع. وكذلك فإن قيمة نوع مهمة تحدد مهمة.

Type النوع: يحدد النوع كلاً من مجموعة القيم، ومجموعة العمليات المطبقة على هذه القيم. وتعريف النوع، هو أداة لغة تحدد نوعاً. والنوع، هو نوع وصول، أو نوع مصفوفة، أو نوع خاص، أو نوع تسجيلة، أو نوع سلمي، أو نوع مهمة.

Use Clause عبارة الـ Use: ينجز هذا السياق الرؤية المباشرة للتصريحات التي تظهر في الأجزاء المرئية للحزم البرمجية المسمّاة.

Variable المتحول: انظر .Object

Variante Part القسم المتبدل: يحدد القسم المتبدل لتسجيلة خيارات عناصر التسجيلة المتعلقة بمميز التسجيلة، وكل قيمة لمميز تؤسس خياراً خاصاً لجزء المحول.

Visibility الرؤية: في مكان ما من البرنامج يكون تصريح كيان بمعرف ما مرئياً، إذا كان الكيان يمثل مدلولاً مقبولاً من أجل حالة لمعرف في تلك النقطة. والتصريح يكون مرئياً بالاختيار في مكان الناخب للعنصر المنتخب، أو في مكان الاسم في الارتباط الاسمي. في غير ذلك، يكون التصريح مرئياً مباشرة إذا كان للمعرف وحده هذا المدلول.

Visible Part القسم المرئي: انظر .Package

With Clause عبارة :with انظر وحدة الترجمة .Compilation Unit.

المراجع

Bibliography

Software Engineering with ADA, Grady BOOCH and Doug BRAYAN , Third Edition 1994

Ingeniere du Logiciel avec ADA De la conception a la realisation, Texte francais de Jean – Pierre ROSEN, ENST 1991

عناوين صدرت في سلسلة الرضا للمعلومات

اسم الكتاب	المؤلف	تاريخ النشر
١- بيئة النوافذ WINDOWS 3.11	م. أحمد شريك	١٩٩٤
٢- مبادئ الصيانة والشبكات	م. عبد الله أحمد	١٩٩٤
٣- معالجة النصوص MS WORD 6.0	د. هيثم البيطار	١٩٩٥
٤- ادخل إلى عالم WINDOWS 95	م. مهيب النقري	١٩٩٦
٥- قواعد البيانات MS ACCESS	زياد كمرجي - بيداء الزير	١٩٩٧
٦- توابع وماكروات في MS EXCEL 97	أ. زياد كمرجي	١٩٩٧
٧- مرجع تعليمي شامل لبرنامج معالجة النصوص MS WORD 97	د. هيثم البيطار	١٩٩٧
٨- مرجع تعليمي شامل في MS EXCEL 97	أ. زياد كمرجي	١٩٩٧
٩- مرجع تعليمي شامل في صيانة الحواسب الشخصية	م. عبد الله أحمد	١٩٩٨
١٠- مرجع تعليمي في برنامج الرسم والتصميم الهندسي AUTOCAD 14	م. احسان مردود	١٩٩٨
١١- المرجع التدريبي الشامل لـ WINDOWS 98	م. إياد زوكار	١٩٩٨
١٢- ادخل إلى عالم WINDOWS 98	م. مهيب فواز النقري	١٩٩٨
١٣- الإنترنيت وإنترانيت وتصميم المواقع	م. عبد الله أحمد	١٩٩٨
١٤- تكنولوجيا المعلومات على أعتاب القرن الحادي والعشرين	هاني شحادة الخوري	١٩٩٨

العنوان	المؤلف	السنة
١٥-الإدارة الاستراتيجية للشركات والمؤسسات	د.يونس حيدر	١٩٩٩
١٦-نظام الـ ISO 9004-1	م.محمد حسن -م.بسام عزام	١٩٩٩
١٧-القائد المفكر حافظ الأسد والمشروع التنموي الحضاري	د.رياض عواد-أ.هاني الخوري	١٩٩٩
١٨- فن إدارة البشر	د. محمد مرعي مرعي	١٩٩٩
١٩- المرجع الشامل لتعليمات برنامج AUTOCAD	م. احسان المردود -م. وهبي معاد	١٩٩٩
٢٠- الدعاية والتسويق ومعاملة الزبائن	م. حنا بللوز	١٩٩٩
٢١- المعلومياء (المعلوماتية) ظروفها وآثارها الاقتصادية - الاجتماعية	د. معن النقري	١٩٩٩
٢٢- المرجع الشامل لبرنامج 3D STUDIO MAX - الجزء الأول	م. جورج عطا لله بركات	١٩٩٩
٢٣- دليل الجودة في المؤسسات والشركات	د. طلال عبود-أ.ماهر العجي	١٩٩٩
٢٤-المرجع المفيد في علم شبكات الحواسيب	د.معتصم شفا عمري	١٩٩٩
٢٥- ادخل إلى عالم ORACLE 8	م. مهيب النقري	١٩٩٩
٢٦- أسس إدارة الموارد البشرية	د. محمد مرعي مرعي	١٩٩٩
٢٧- تعلم برنامج إدارة قواعد البيانات	أ. زياد كمرجي - م. مهيب النقري	١٩٩٩
٢٨- الدليل الشامل لأساسيات الحاسوب والمعلوماتية	م. عبد الله أحمد	١٩٩٩
٢٩- الكذبات العشر للعولمة	د. عدنان سليمان	١٩٩٩
٣٠- بعض مسائل الاقتصاد اللاسياسي	د. مطانيوس حبيب	١٩٩٩
٣١- دليل إعادة تنظيم المؤسسات	د. محمد مرعي مرعي	١٩٩٩

٤٨- مرجع أساسيات الحوسبة	إشراف م.قاسم شعبان- شادي سيدا	٢٠٠٠
الجزء الأول: أساسيات الحاسوب		
٤٩- دليل المديرين في إدارة الأفراد		
وفرق العمل	د. محمد مرعي مرعي	٢٠٠٠
٥٠- بناء التطبيقات باستخدام		
Oracle Developer	م. مهيب النقري	٢٠٠٠
٥١- فن وعلم إدارة الوقت	أ. رعد الصرن	٢٠٠٠
٥٢- الأخلاق الحديثة للإدارة		
الإدارة بالقيم	د. عدنان سليمان	٢٠٠٠
٥٣- من الفكرة إلى المنتج - إدارة الإبداع	د. حسين علي	٢٠٠٠
٥٤- دليل المطورين إلى دلفي Delphi	م. حسن شاليش حسن -	
	م. سامر سعيد- م. ميشيل الياس	٢٠٠٠
٥٥-المعالجات التحكمية	م. عبده هلالة	٢٠٠٠
٥٦- الدليل العملي لتطبيق		
نظام الـ HACCP	م. ماهر العجي - م. ميلاد عربش	٢٠٠٠
٥٧- EXCEL 2000 - الجزء الأول	م. إياد زوكار- م. محمد الضمّاد	٢٠٠٠
٥٨- أساسيات الانترنت	د. ماهر سليمان-	
	م. حسام عابد - م. إياد خدّام	٢٠٠٠
٥٩-الانترانت - بنيتها الأساسية		
وانعكاساتها على الشركات	د. عمار خير بك - م. حسام اللحم	٢٠٠٠
٦٠- البحث عن المعلومات في الإنترنت	د. عمّار خير بك	٢٠٠٠
٦١- التسويق عبر الانترنت	د. طلال عبود	٢٠٠٠
٦٢- الحساسات وطرق الربط		
إلى أنظمة التحكم المبرمج	م. عبده هلاله - م. عامر عبود	٢٠٠٠
٦٣- المدخل إلى نظام		
Windows NT 4 Server	م. احسان مردود	٢٠٠٠

٦٤- أساسيات الحوسبة - الجزء الثاني	م. قاسم شعبان	٢٠٠٠
٦٥- دليل التحفيز في المؤسسات والإدارات	د. محمد مرعي مرعي	٢٠٠٠
٦٦- دليل التغيير في المؤسسات والإدارات	د. محمد مرعي مرعي	٢٠٠٠
٦٧- اقتصاديات النقود والصيرفة في سوريا	د. علي كنعان	٢٠٠٠
٦٨- تقنية المعلومات في إدارة الشركات	م. قاسم شعبان	٢٠٠٠
٦٩- إدارة الابتكار والابداع	أ. رعد الصرن	٢٠٠٠
٧٠- ٧٩- سلسلة الرضا لتبسيط علوم الحاسوب	م. مهيب النقري - د. معتصم شفا عمري	٢٠٠٠
٨٠- أساسيات الإدارة المالية الحديثة - ج٢	د. دريد درغام	٢٠٠٠
٨١- الاتصال والاتصال الإداري	د. سامر جلعوط	٢٠٠٠
٨٢- مهارات البيع	د. حسين علي	٢٠٠٠
٨٣- أساسيات Windows 2000	م. مهيب النقري	٢٠٠٠
٨٤- المرجع الأساسي في Macromedia Director 7	أ. وائل جلال	٢٠٠٠
٨٥- أساسيات التجارة العالمية - ج ١	أ. رعد الصرن	٢٠٠٠
٨٦- التحريك في برنامج 3D Max - الجزء الثالث	م. جورج بركات	٢٠٠٠

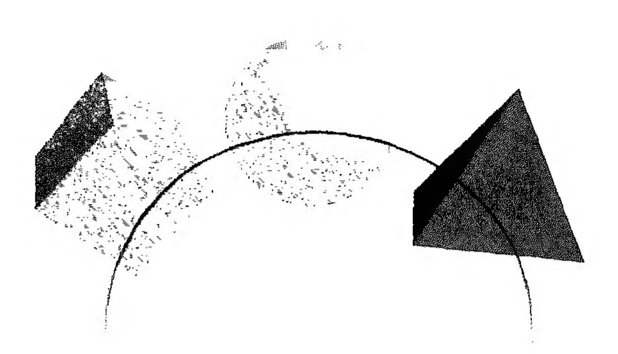
عناوين ستصدر قريباً

اسم الكتاب	المؤلف	تاريخ النشر المتوقع
١- دليل التطوير الإداري والحصيلة الاجتماعية	د. محمد مرعي مرعي	٢٠٠٠
٢- تصميم المواقع WEB DESIGN	م.عبد الله أحمد	٢٠٠٠
٣- المعلوماتية الطبية	د.نبيل دك الباب	٢٠٠٠
٤- كتاب Autocad 2000	م. احسان مردود - م. وهبي معاد	٢٠٠٠
٥- المرجع الأساسي في Macromedia Flash 5	أ. وائل جلال	٢٠٠٠
٦- نظام Windows 2000 Server	د. صلاح دوه جي - م. مهيب النقري	٢٠٠٠
٧- برنامج Sap 2000	م. أيمن عابد	٢٠٠٠
٨- برنامج معالجة الصور Adobe Photoshop 5.5	م. جورج بركات	٢٠٠٠
٩- الحاسوب في عالم التحكم	م. عبده هلاله - م. عامر عبود	٢٠٠٠
١٠- سلسلة الرضا للبرامج الهندسية التطبيقية	م. عبده هلاله-م. عامر عبود	٢٠٠٠
١١- لغة جافا سكريبت	م. حسام أسعد - د. عمار خير بك	٢٠٠٠




Converted by Tiff Combine - (no stamps are applied by registered version)

متقدم متوسط مبتدىء



ظهرت علامات أزمة البرمجيات في السبعينات في الولايات المتحدة الأمريكية على شكل برمجيات لا توافق حاجات المستخدمين، قليلة الوثوقية، كثيرة الكلفة، غير مرنة، صعبة الصيانة وغالباً ما يتأخر البرمجي ضمن الزمن المسموح به وأصبحت البرمجيات أكثر ضخامة وتعقيداً. فقاوموا الأزمة باستعمال العديد من الأدوات البرمجية مثل تقنيات البرمجة البنيوية وطرق التصميم الغرضية التوجه حيث تعتمد هذه الأدوات على مجموعة من المفاهيم الأسياسة .

وكرد فعل على أزمة البرمجيات مولت وزارة الدفاع الأمريكية تطوير لغة برمجة قوية جداً أصبحت لغت **ADA**

الدافع الأساسي الذي جعلني أقوم بترجمة كتاب هندسة البرمجيات باستخدام أدا :

Software Engineering with ADA

للمؤلفين Doug Brayan and Grady Booch هو أني عايشت **ADA** في مراحل ولادتها عندما حضرت الدكتوراة (١٩٧٨ - ١٩٨٢) حيث كانت جزءاً من موضوع أطروحتي. وقد وجدت في هذا الكتاب ما يبرز أهمية هذه اللغة من حيث عموميتها وقدرتها العالية على التعبير لتكون متخصصة في جميع المجالات المعقدة العلمية منها والإدارية ودافعاً من إحساسي بأنه من واجبي إغناء مكتبتنا العربية وتعريف المهتمين العرب في مجال المعلوميات بهذه اللغة .

تعتبر أدا واحدة من اللغات الحديثة جداً ذات قدرة تعبير عالية وذات استخدام عام وقد صممت من قبل فريق دولي لتلبي جميع المتطلبات البرمجية لوزارة الدفاع الأمريكية **DOD** رداً على أزمة تطوير البرمجيات وقد تم التوصل إليها بعد دراسات وتحاليل طويلة دامت أكثر من أربع سنوات تم بنتيجتها تصميم وتعريف لغة **ADA** واعتماد معيار خاص بها عام ١٩٨٢ .

وقد صممت خصيصاً في مجال الأنظمة المعلوماتية الضخمة في الزمن الحقيقي، والمحمولة التي تتطلب وثوقية عالية كما أنها برعت في مجالات علمية معقدة .

